Export de NetFlows 9 sous Android et Inférence de la localisation d'un utilisateur d'ordiphone sans données géotagguées

Julien Vaubourg

Stage ingénierie & recherche
3e année de TELECOM Nancy

Présenté par

Julien VAUBOURG
julien@vaubourg.com

Export de NetFlows 9 sous Android et Inférence de la localisation d'un utilisateur d'ordiphone sans données géotaggées

Encadré par

Abdelkader LAHMADI & Moufida MAIMOUR
abdelkader.lahmadi@loria.fr - moufida.maimour@loria.fr

Au sein de

Équipe Madynes - LORIA/INRIA Grand-Est

Juillet 2013

Préface

Ce document constitue le rapport de stage d'un étudiant de dernière année de TELECOM Nancy.

Il s'est déroulé du 1^{er} avril 2013 au 31 juillet 2013 au sein de l'équipe Madynes (Inria) du LORIA. Encadré par Abdelkader LAHMADI (Madynes) et Moufida MAIMOUR (TELECOM Nancy), il a pour objectif de valider des compétences d'ingénieur, tout en offrant une première approche concrète du domaine de la recherche.

Remerciements

Je tiens particulièrement à remercier :

– Abdelkader LAHMADI, pour son encadrement et les discussions que nous avons pu avoir durant ce stage ;

– Alexandre BOEGLIN, pour son aide et sa disponibilité ;

– Céline SIMON et Julie GRZESIAK, pour leur accueil et leur dynamisme ;

– Olivier FESTOR, pour m'avoir accueilli au sein de son équipe et m'avoir orienté pour ma poursuite en doctorat ;

– et par ordre alphabétique : Martín BARRÈRE, César BERNARDINI, Narjess DEROUICHE, François DESPAUX, Éric FINICKEL, Gaëtan HUREL, Anthéa MAYZAUD, Kévin ROUSSEL, Mohamed Saïd SEDDIKI et Juan Pablo TIMPANARO qui ont été les stagiaires et doctorants qui m'ont sympathiquement accompagnés durant ces quelques mois et auxquels je souhaite le meilleur pour la suite.

Table des matières

Table des figures

Introduction

Contexte du stage

Ce stage clôture mon parcours TELECOM Nancy, et intervient à l'issue de la troisième et dernière année. Il constitue la dernière étape pour la validation du diplôme d'ingénieur en informatique, grâce à une ultime expérience dans le monde professionnel sur une période de quatre mois.

Ayant fait le choix de continuer en doctorat dans la mesure du possible, ce stage est orienté recherche et a pour second objectif de valider une expérience dans ce domaine. Il est encadré par Abdelkader LAHMADI, enseignant-chercheur au sein de l'équipe Madynes du laboratoire lorrain de recherche en informatique et ses applications. Il se déroule au sein de la structure Inria et est encadré universitairement par Moufida MAIMOUR, enseignante-chercheuse à TELECOM Nancy.

Après une courte présentation de l'école, de Inria et du laboratoire, ce rapport synthétise le travail effectué durant ce stage. Ce dernier comprend une partie ingénierie suivie d'une partie purement recherche, les deux parties convergeant vers un même but. La première partie aura consisté à développer un exportateur de NetFlows modulaire pour Android et la seconde partie à écrire d'un papier de recherche sur l'inférence de la localisation d'un utilisateur d'ordiphone (*smartphone*) sans données géotagguées. Chaque partie ayant conduit à la production d'un résultat concret, une dernière partie combine ces deux résultats pour présenter un résultat final qui devrait faire l'objet de présentations dans différentes conférences.

TELECOM Nancy

TELECOM Nancy, école associée de l'Institut Mines-Télécom, est une école d'ingénieurs publique de l'Université de Lorraine. Il s'agit d'une école généraliste en informatique, sciences et technologies du numérique, habilitée par la Commission des Titres d'Ingénieurs (CTI). Elle fait partie du Concours TELECOM INT, et a changé d'identité au début de l'année 2012, se faisant connaître jusqu'alors sous le nom de ESIAL.

En plus d'un tronc commun, elle dispose de quatre spécialisations différentes, qui vise quatre domaines distincts :

Ingénierie du Logiciel Développement d'applications sûres et robustes, avec un attachement particulier pour le domaine du web.

Logiciels Embarqués Étude des systèmes embarqués et programmation efficace dans des environnements restreints.

Systèmes d'Information d'Entreprise Étude des systèmes d'information et des bases de données.

Télécommunications, Réseaux et Services Étude du fonctionnement et de la gestion des réseaux informatiques.

Le parcours est régulièrement complété par des stages au fil des trois ans, avec un stage plus long en fin d'études. C'est dans le cadre de la dernière année que ce stage se déroule, au sein de l'équipe Madynes.

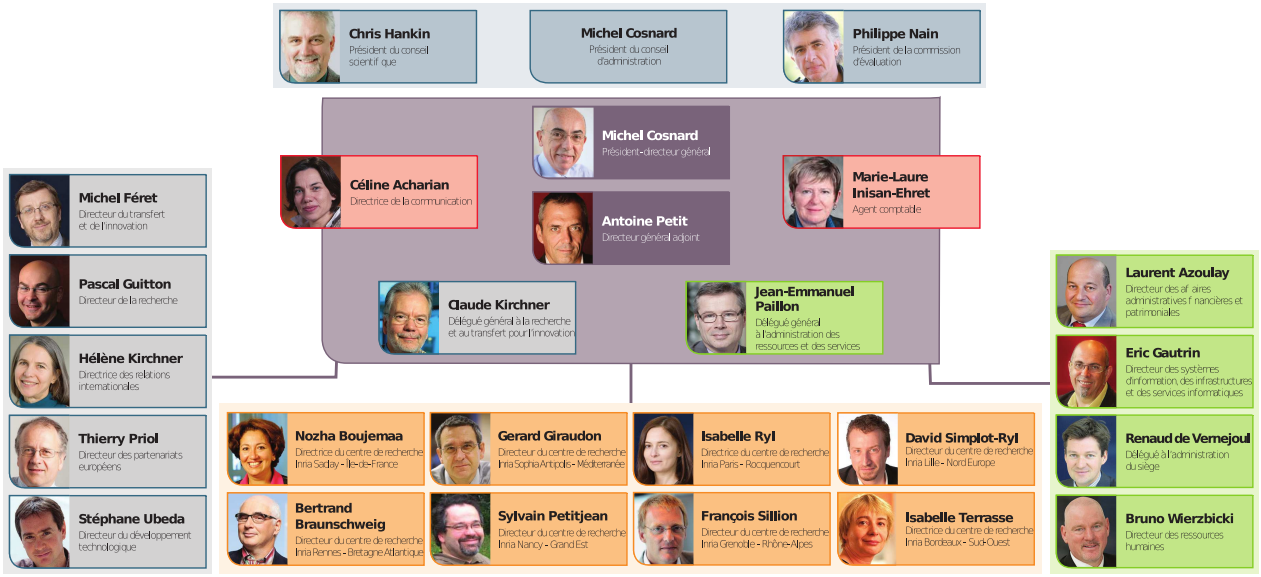

FIGURE 1 – Organigramme de Inria.

Madynes

L'équipe Madynes est une équipe de recherche du laboratoire LORIA, lui-même rattaché à Inria Grand Est, composante de Inria.

Inria

Inria est un établissement public à caractère scientifique et technologique créé le 3 janvier 1967. Son objectif est de mettre en réseau les compétences et talents de l'ensemble du dispositif de recherche français dans le domaine des sciences et technologies de l'information et de la communication.

Inria est composé de huit centres de recherche autonomes dont celui de Nancy Grand Est, pour lequel ce stage a été effectué.

Quelques chiffres permettent de situer l'importance de la structure dans la recherche française :

- 4290 employés ;
- 3429 scientifiques ;
- 208 équipes de recherche ;
- 171 équipes-projets ;
- 4850 publications ;
- +290 thèses soutenues ;
- 271 brevets actifs ;
- 111 logiciels déposés à l'agence pour la protection des programmes ;
- 105 sociétés de technologie créées ;
- 252 M€ de budget.

L'organisation d'Inria se compose de :

- une équipe de direction constituée du président-directeur général, du directeur général adjoint, du délégué général à l'administration des ressources et des services ainsi que des directeurs des huit centres Inria ;
- un délégué général à la recherche et au transfert pour l'innovation ;
- un délégué général à l'administration des ressources et des services ;
- une instance administrative, deux instances scientifiques et un comité externe.

Un organigramme complet est proposé en figure 1.

LORIA et Inria Grand Est

Le LORIA (*Laboratoire Lorrain de Recherche en Informatique et ses Applications*) est une UMR (*Unité Mixte de Recherche*) commune à plusieurs établissements :

- CNRS (*Centre National de Recherche Scientifique*) ;
- Université de Lorraine ;
- Inria.

Avec un effectif de plus de quatre cent cinquante employés, le LORIA accueille cent cinquante chercheurs et enseignants-chercheurs, des doctorants et post-doctorants, des ingénieurs, des techniciens et des personnels administratifs. Ils sont organisés en équipes de recherche et services de soutien à la recherche. Le laboratoire est organisé par une équipe de direction, un conseil de laboratoire, une assemblée des responsables d'équipes, un comité d'hygiène et sécurité ainsi que des commissions des utilisateurs des moyens informatiques, de la documentation et de la formation.

Ses principales missions sont la recherche fondamentale et appliquée dans les domaines des sciences et technologies de l'information et de la communication, la formation par la recherche, et le transfert technologique par le biais de partenariats industriels et par l'aide à la création d'entreprises. Les thèmes récurrents de recherche sont les calculs, la simulation et visualisation à hautes performances, la qualité et la sûreté des logiciels, les systèmes parallèles, distribués et communicants, les modèles et algorithmes pour les sciences du vivant, le traitement de la langue maternelle et la communication multimodale ainsi que la représentation et la gestion des connaissances.

Les principaux domaines d'application sont donc les réseaux, Internet et la toile, la sécurité des systèmes informatiques, la réalité virtuelle, la robotique, la bioinformatique et enfin la santé.

L'équipe Madynes

L'équipe Madynes (*Management of Dynamic Networks and Services*) a pour thématique la recherche, la conception, la validation et la mise en œuvre de méthodes de supervision et de contrôle adaptés aux réseaux et aux services dynamiques émergents. Elle s'intéresse plus particulièrement aux politiques de sécurité des réseaux, la voix sur IP, le protocole IPv6, les réseaux pair-à-pair, et plus récemment la sécurité sur Android.

Cette équipe est sous la direction de Olivier FESTOR, également directeur de TELECOM Nancy. Elle est composée de plus de vingt personnes, dont des enseignants-chercheurs, des doctorants, des ingénieurs, une assistante de projet et des étudiants stagiaires.

Chapitre 1

Mise en œuvre d'une sonde NetFlow pour Android

Sommaire

Cette section présente le travail d'ingénierie, qui a consisté à écrire un exportateur de NetFlows modulaire pour les ordiphones Android.

1.1 Android

Android est un système d'exploitation basé sur le noyau Linux, initialement destiné aux ordiphones et aux tablettes numériques à base d'écrans tactiles. Il s'agit d'un produit développé par la société Google qui a racheté la jeune pousse éponyme en 2005.

Son code source est libre (licence Apache), et peut ainsi être lu, modifié et redistribué sans contrainte. Il dispose à ce titre d'une large communauté d'utilisateurs et de développeurs, qui participent activement à son évolution et son amélioration.

Grâce à son indépendance vis-à-vis du matériel (au contraire de l'iPhone de Apple), il est devenu la plateforme mobile la plus utilisée dans le monde. Il dispose d'un espace centralisé géré par Google, permettant d'installer automatiquement des applications proposées par la communauté. Il n'existe quasiment aucune modération sur ces applications, qui peuvent donc être corrompues et malveillantes. Il existe beaucoup de travaux de recherche destinés à parfaire la sécurité de ce système en faisant des contrôles a priori ou a posteriori.

La plateforme Android a été choisie dans le cadre de ce stage pour sa popularité et son ouverture, facilitant l'écriture de programmes complexes.

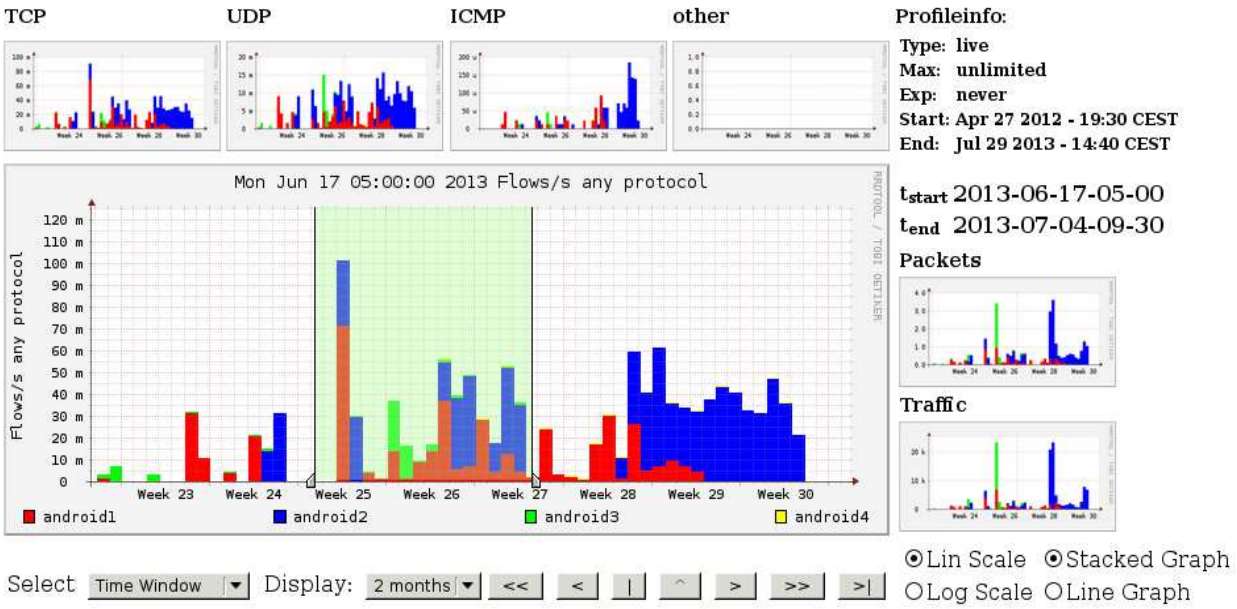

FIGURE 1.1 – Exemple de collecteur (Nfsen basé sur Nfdump).

1.2 Protocole NetFlow version 9

1.2.1 Propriétés

Le protocole NetFlow est conçu en 1996 par la société Cisco, avec l'objectif de fournir aux administrateurs réseau des informations sur leur trafic réseau. Les données sont récupérées directement sur les équipements réseau (commutateurs ou routeurs) et exportées vers un collecteur externe. Elles sont destinées à synthétiser le trafic réseau, permettant aux administrateurs de superviser son activité, dans un but de sécurisation, d'optimisation ou de facturation (voir l'exemple de collecteur en figure 1.1).

Un flux est défini comme une séquence undirectionnelle de paquets avec des propriétés communes lisibles depuis un équipement réseau. Ces propriétés peuvent être très variées, en allant des adresses IP aux ports applicatifs en passant par des métriques d'accumulation comme le nombre d'octets transmis durant le transfert. Grâce à la version neuf du protocole qui a été utilisée pour ce travail, un système de modèles (*templates*) puissant permet d'ajouter n'importe quel type d'information dans les NetFlows transmis.

L'approche du protocole par modèles permet plusieurs avantages notables :

- de nouveaux champs peuvent être ajoutés dynamiquement aux NetFlows sans changer la structure du format du paquet d'export. En cas de changement dans la liste des champs, les anciennes versions imposaient de redéfinir statiquement le format d'export et de modifier le collecteur distant pour qu'il supporte le nouveau format et soit en capacité d'extraire les informations utiles ;

- les modèles sont envoyés de la même façon que les données, dans des paquets d'export de NetFlows. Ainsi, même si le collecteur distant ne comprend pas la signification d'un champ, il connait au moins la taille qu'il est censé représenter et peut donc accéder aux champs qui suivent ;

- un modèle représentant une liste de champs associés à leur taille, et plusieurs modèles pouvant être utilisés en même temps, ce système permet d'optimiser les envois au collecteur. Ainsi, si un champ particulièrement imposant n'a pas à être utilisé pour tous les types de flux (par exemple une adresse IPv6 ou un nom d'application), il est possible de définir deux modèles distincts et d'utiliser l'un ou l'autre pour éviter de transmettre des champs vides qui doivent malgré tout être envoyés.

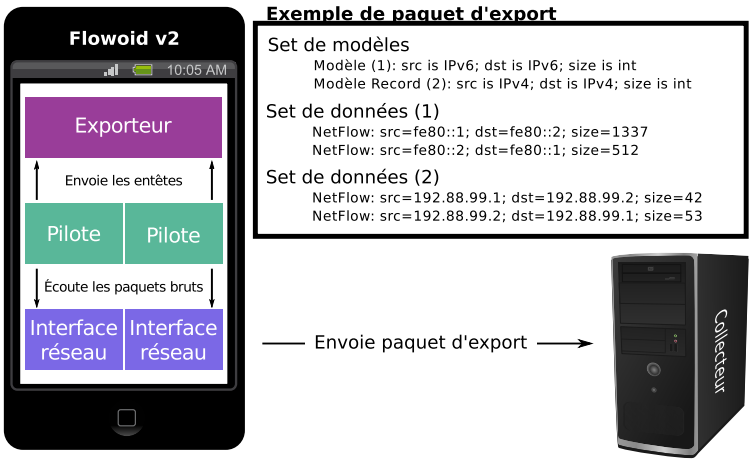

FIGURE 1.2 – Exemple de paquet d'export.

1.2.2 Terminologie

Le protocole NetFlow impose un vocabulaire relativement important, nécessaire à la bonne compréhension de son fonctionnement.

Les NetFlows sont envoyés par l'exportateur (dans notre cas un ordiphone Android) vers le collecteur (un serveur distant). Le conteneur global des NetFlows qui voyage est appelé un paquet d'export.

Chaque paquet d'export contient un ou plusieurs sets de données. Il peut également contenir un set de modèles, qui sera analysé et enregistré par le collecteur pour pouvoir interpréter les données reçues. Le set de modèles devrait toujours être envoyé avant l'envoi des premières données, et régulièrement retransmis de façon à mettre à jour le collecteur. Lorsqu'un lot de NetFlows est transmis dans un paquet d'export, chaque set de données correspond à un modèle précis qui est décrit dans le set des modèles.

Chaque set est constitué d'enregistrements. Ainsi, un set de modèles contiendra une liste d'enregistrements de modèles et un set de données contiendra une liste d'enregistrements de flux réseaux (NetFlows). Un enregistrement de modèle est composé d'un identifiant et d'une longueur correspondant à la taille fixe que représentera sa donnée. Les identifiants sont standardisés par des définitions de type de champ [RFC 3954 section 8], mais de nouveaux types peuvent être ajoutés, en s'assurant que le collecteur saura les interpréter (par défaut il les ignorera tout en étant capable de lire les suivants). Les enregistrements de flux rangés par modèle (partageant donc tous la même syntaxe) dans les sets de données consistent simplement en des suites de données de taille fixe ordonnées selon la définition du modèle associé.

La figure 1.2 présente un exemple de paquet d'export, à partir de l'exportateur Flowoid qui sera présenté dans la section suivante.

Les sets de type option qui sont décrits dans la RFC 3954 ne seront pas détaillés ici puisqu'ils n'ont pas été utilisés dans ce travail.

1.2.3 IPFIX

Alors que le protocole NetFlow est un standard imposé et détenu par Cisco plutôt qu'une norme (RFC informationnelle), des contributeurs de l'IETF ont proposé une alternative sous le nom de IPFIX (*IP Flow Information Export*) avec la RFC 3917.

Le protocole IPFIX est très fortement influencé par la version neuf de NetFlow qui a été décrite plus haut, notamment en reprenant son système de modèles. Hormis quelques variations dans la terminologie, il apporte quelques nouveautés comme des solutions en terme de sécurité.

Une RFC en brouillon [1] sur l'ajout de champs destinés à accueillir des données de géolocalisation est à retenir. Une fois ce protocole normalisé et le brouillon sur les données de géolocalisation acceptées, l'exportateur de NetFlows proposé dans la section suivante pourra être adapté pour passer facilement de NetFlow en version neuf à IPFIX. Ce brouillon est co-écrit par Abdelkader LAHMADI et Olivier FESTOR.

1.3 La sonde Flowoid

1.3.1 Exporteur de NetFlows pour Android

Le protocole NetFlow (largement éprouvé et adopté) a naturellement été retenu pour les besoins du stage. Si le côté collecteur est un besoin standard dès lors qu'on travaille avec des NetFlows, le côté export se situe dans notre cas dans un environnement mobile. Comme indiqué plus haut, c'est la plateforme Android qui a été retenue pour mener les expérimentations.

L'étude qui sera menée dans la seconde partie de ce rapport nécessite de travailler à partir des flux qui sont à la disposition de n'importe quel fournisseur d'accès à Internet mobile. Dans la mesure où nous ne pouvons pas avoir accès à ce type d'équipement, les flux doivent obligatoirement être récupérés directement depuis le périphérique à l'aide d'un exportateur intégré qui communique directement avec le collecteur distant. Puisqu'il n'existe pas d'exportateur de NetFlows performant pour Android, le développement de celui-ci a naturellement constitué la partie ingénierie du stage.

Grâce à la version neuf du protocole et son puissant système de modèles, l'exportateur doit retranscrire cette souplesse dans son implémentation. Ainsi, une programmation la plus modulaire possible a été choisie pour créer cet exportateur qui doit être le plus évolutif possible.

1.3.2 Bibliothèque abstraite

Le travail consiste en deux principales parties :

1. la brique logicielle qui crée les paquets d'export en respectant la RFC 3954 ;
2. l'interface utilisateur qui permet d'accéder aux différents paramètres à positionner.

En respectant le modèle de conception modèle-vue-contrôleur (MVC), la première partie correspondrait au modèle, et la seconde à la vue avec les contrôleurs qui permettent de modifier le modèle.

Plutôt que de ne proposer qu'une application complète avec des choix d'implémentation arbitraires qui ne correspondraient pas forcément à tous les usages, nous avons décidé de clairement séparer les deux en créant une bibliothèque Java pour Android. Cette dernière est abstraite (elle ne peut pas être utilisée sans une implémentation de ses principales classes) et permet d'implémenter très facilement un exportateur de NetFlows pour Android.

Puisqu'il s'agit avant tout de l'implémentation d'une RFC très détaillée, la bibliothèque se veut le plus proche possible du standard, pour être elle-même le plus standard possible.

Les modèles de la version neuf de NetFlow offrant une flexibilité des formats d'export, la bibliothèque permet d'implémenter très facilement de nouveaux types de données à ajouter aux flux exportés. Nous verrons précisément de quelle façon en étudiant l'implémentation qui en a été faite pour Flowoid.

La figure 1.3 donne une vision épurée des classes et méthodes qu'elle offre aux développeurs. Les classes en italique sont abstraites, ce qui est le cas des classes les plus importantes. On note également la classe `Driver`, qui reflète un mode de fonctionnement singulier de l'application qui sera étudié par la suite.

1. http://tools.ietf.org/html/draft-festor-ipfix-metering-process-location-01

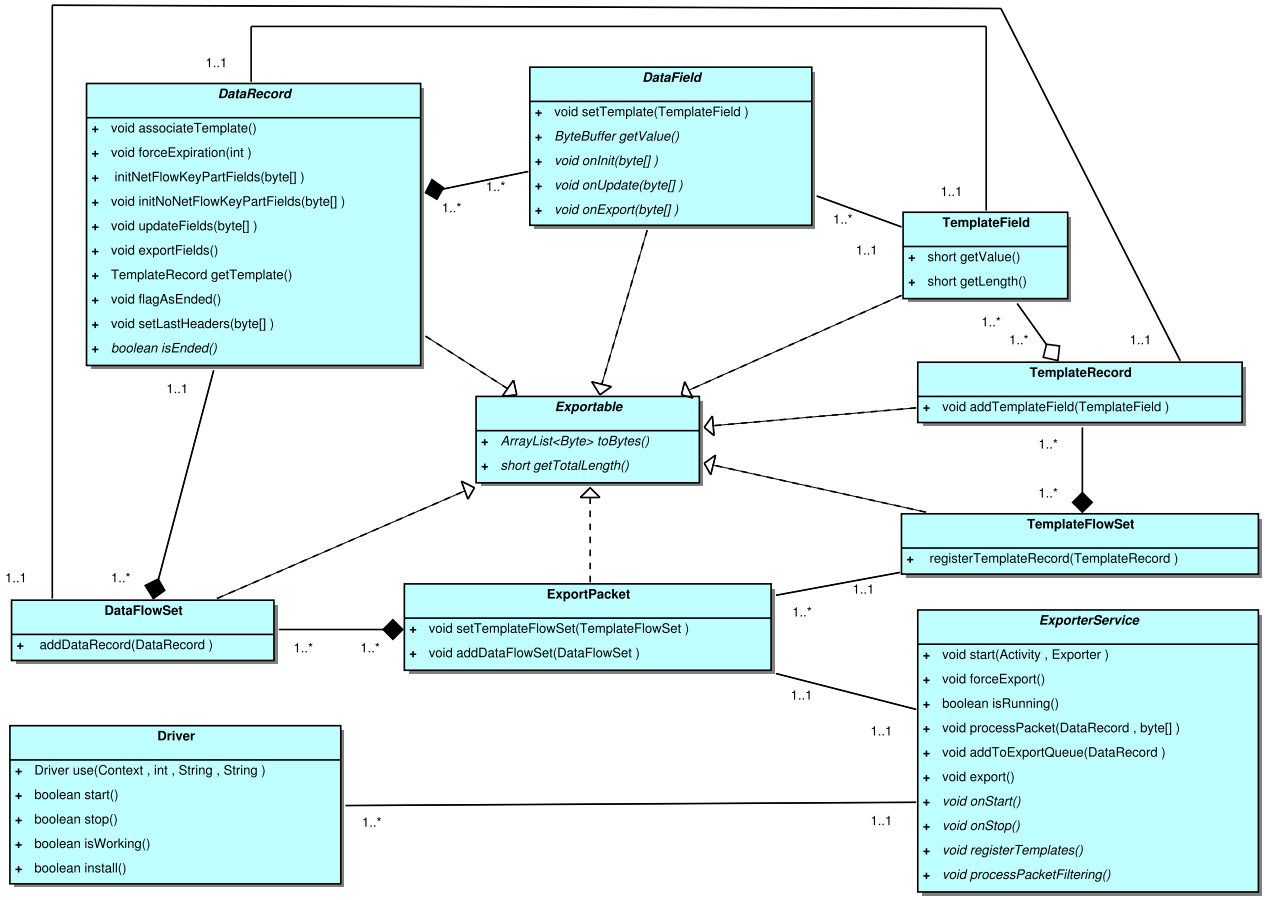

FIGURE 1.3 – Diagramme UML de la bibliothèque FlowoidExporter.

1.3.3 Pilotes de communication

Android est une distribution GNU/Linux qui utilise principalement une machine virtuelle Java appelée Dalvik. Toutes les applications crées pour Android sont des programmes Java exécutés dans leur machine virtuelle optimisée pour mieux contrôler les différents accès aux informations du système. Cette rigueur apporte des avantages, comme une interface de programmation (API) particulièrement robuste et complète. Ainsi, depuis un programme Java sur Dalvik, il est possible d'accéder facilement à des informations comme les coordonnées géographiques, le niveau de la batterie, etc.

Mais un exportateur de NetFlows, dans sa forme la plus simple, consiste à écouter les paquets qui transitent sur l'interface réseau du périphérique pour les synthétiser et les renvoyer sous une autre forme. Écouter le réseau n'est pas une opération anodine, puisqu'elle permet de récupérer toutes les informations que font transiter toutes les autres applications du système. Dans un environnement GNU/Linux, il s'agit d'un privilège qui n'est accordé qu'à l'utilisateur suprême du système, communément appelé le super-utilisateur.

Contrairement à un système GNU/Linux sur machine de bureau, les utilisateurs d'Android ont rarement accès aux plus haut privilèges de leur propre matériel. Il est toutefois souvent possible de les récupérer en rendant l'accès au super-utilisateur possible. Cette opération est souvent jugée illégale par les opérateurs qui rendent parfois la garantie caduque dès lors qu'on se permet de vouloir contrôler son propre appareil. Cette réticence est souvent justifiée par l'installation d'applications par défaut liées à des partenariats commerciaux de l'opérateur, qui ne souhaite donc pas qu'elles puissent être retirées. Cette contrainte limite les possibilités de déploiement de Flowoid.

Si les téléphones qui sont directement distribués par Google ne souffrent pas de cet abus, ce dernier n'a pour autant pas prévu l'utilisation de Dalvik et son API directement avec le super-utilisateur. L'astuce est donc de recourir directement à une commande système via Java, pour lancer un binaire en prenant soin de de l'exécuter en super-utilisateur. On parle alors de code natif,

puisque ce binaire doit de préférence être écrit en C. Il n'a donc pas accès à l'API de Dalvik et ses nombreuses possibilités, ce qui aurait rendu trop fastidieux l'écriture complète de l'exportateur en C.

C'est pour cette raison qu'intervient la notion de pilote (*driver*) qui est visible dans la figure 1.2. Celui-ci est lancé avec les droits de super-utilisateur directement sur le système, et se charge d'écouter les interfaces réseau grâce à la bibliothèque *pcap* (utilisée dans des outils d'administration réseau très connus, tels que Wireshark ou Tcpdump). Le choix a été fait de réduire au minimum l'intelligence du pilote pour qu'il ne soit qu'une interface entre le réseau et le programme Java. Ainsi, il se contente d'écouter et de récupérer les paquets, pour en extraire les entêtes jusqu'à la couche transport. Via une *socket* interne, il transmet en temps réel les entêtes à l'interface Java qui peut les traiter en exploitant toute la souplesse du Java et toute la puissance de l'API Android.

À noter qu'il existe une interface native pour Java (JNI) qui peut être aussi utilisée sur Dalvik et qui aurait pu correspondre à cet usage, en permettant une communication directe entre le programme Java et le programme C, sans passer par un mode de communication réseau. Après divers essais, Android est suffisamment bien conçu pour que les programmes C natifs lancés de cette façon soient exécutés avec l'utilisateur aux droits limités qui est attribué à l'application. Ainsi, il est impossible de passer en super-utilisateur et donc d'écouter les interfaces en utilisant cette technique.

Les pilotes sont spécifiques à un type de paquet en particulier. Dans le cas de l'exportateur de NetFlows implémenté — qui sera présenté par la suite — le pilote pour IP a été entièrement développé, mais la bibliothèque de l'exportateur a été écrite de façon suffisamment modulaire pour que tout autre type de paquets provenant d'autres types d'interfaces puissent être transmis à l'application Java. On pourrait par exemple écouter sur une interface bluetooth et transmettre les paquets écoutés. Il suffit que le côté Java ait connaissance de la nature du pilote, pour utiliser le parseur adéquat qui permettra de traiter les entêtes reçus et les transformer en NetFlow avec les modèles appropriés pour le collecteur. Plusieurs pilotes peuvent être utilisés en même temps, et envoyés au même collecteur grâce à la pluralité des formats que permet d'exporter la version neuf de NetFlow au sein d'un même paquet d'export.

L'implémentation des pilotes natifs fait partie de la phase implémentation de la bibliothèque.

1.3.4 Structure de la bibliothèque

1.3.4.1 Classes

La figure 1.3 indique que la bibliothèque comporte une dizaine de classes.

Pour transmettre les NetFlows, l'exportateur envoie un ou plusieurs `ExportPacket` à un collecteur distant :

- `ExportPacket :` un paquet d'export qui contient un ou plusieurs `DataFlowSet` et parfois un `TemplateFlowSet`.

- `DataFlowSet :` un set de NetFlows qui contient un ou plusieurs `DataRecord`. Chacun d'entre eux est défini avec le même `TemplateRecord` et l'identifiant du `DataFlowSet` correspondra à l'identifiant de ce `TemplateRecord`.

- `DataRecord :` un NetFlow qui contient un ou plusieurs `DataField`.

- `DataField :` un champ qui contient une et une seule valeur du NetFlow (adresse IP, port TCP, date, etc.).

À l'envoi, les paquets d'export peuvent être envoyés en plusieurs fois pour éviter la fragmentation des paquets IP que le collecteur supporte difficilement. Ce dernier saura comment les parser parce que ceux-ci sont régulièrement accompagnés d'un `TemplateFlowSet` parmi les `DataFlowSets` :

– `TemplateFlowSet :` un set de modèles qui contient un ou plusieurs `TemplateRecord`. L'identifiant de ce set est obligatoirement zéro.

– `TemplateRecord :` un modèle qui représente la structure d'un NetFlow, à l'aide d'un ou plusieurs `TemplateField` ordonnés.

– `TemplateField :` un modèle de champ défini grâce à l'un des `FieldTypeDefinition` accessibles via la bibliothèque.

– `FieldTypeDefinition :` un couple identifiant / taille en octets qui indique la sémantique et la longueur d'un champ pour le parseur du collecteur (ils sont décrits dans la section huit de la RFC 3954).

1.3.4.2 Aperçu des spécifications

La plus centrale des classes de la bibliothèque est l'interface `Exportable`, qui assure que toutes ses implémentations seront en capacité d'indiquer le nombre d'octets qu'elle représente, et qu'elles seront capables de renvoyer leur contenu sous forme d'octets bruts.

Puisque quasiment toutes les classes l'implémentent, ses deux fonctions peuvent être calculées de façon récursives. Par exemple, la classe `ExportPacket` calculera sa taille en se contentant de faire l'addition des tailles que lui retourneront tous les `DataFlowSet` qu'elle contient, plus le `TemplateFlowSet`. Concernant les `DataFlowSet`, elles calculeront à leur tour leur taille en additionnant celle de tous les `DataRecord`, qui feront de même pour leurs `DataField`. Enfin, un `DataField` sera en capacité de déterminer sa taille en se référant à la valeur indiquée par le modèle (`TemplateField`) auquel elle est associée.

La plupart des autres classes et méthodes respectent le vocabulaire imposé par la RFC décrite plus haut dans la section 1.2.1, et dont le lien avec les classes est fait dans la section 1.3.4.1. Cette dernière constitue donc une excellente première documentation pour comprendre le fonctionnement de la bibliothèque.

L'implémentation est prévue en deux étapes :

1. les modèles (*template*) à utiliser, notamment grâce à la classe `TemplateField` qui n'est pas abstraite et qui peut donc être directement instanciée en fonction des paramètres qui permettront de créer ses propres modèles ;
2. les données (*data*) qui seront à extraire des paquets, notamment grâce aux classes `DataField` et `DataRecord` qui devront obligatoirement être étendues.

Une classe héritant de `DataField` devra obligatoirement faire référence à un objet de type `TemplateField` lors de son instanciation, pour décrire de quelle façon l'objet est formaté et devra donc être exporté. Diverses vérifications qui peuvent générer des exceptions sont implémentées par la bibliothèque pour interdire autant que faire ce peut les objets `DataField` de proposer des valeurs qui ne correspondent pas à la taille imposée par leur modèle. À titre d'exemple un `DataField` peut représenter une adresse IPv6, qu'il sera capable d'extraire lui-même depuis le paquet brut qui sera passé à l'une de ses fonctions. Lors de l'association d'un `DataField` à un `DataRecord`, le développeur peut préciser s'il s'agit d'un champ faisant partie de l'identité discriminante du NetFlow ou non. C'est par exemple le cas d'une adresse IPv6, qui sera extraire pour le premier paquet et qui ne pourra pas être différente pour les paquets suivants, pour un même NetFlow.

Les classes héritant de `DataRecord` représentent donc les NetFlows. Elles sont obligatoirement associées dès l'instanciation à une série ordonnée de `DataField` qui constitueront la valeur de l'ensemble des champs pour un NetFlow. Puisqu'elles devront également correspondre à un `TemplateRecord` spécifique, la bibliothèque vérifiera si le `TemplateField` des `DataField` associés au fur et à mesure correspondent bien au `TemplateField` de même index associé au `TemplateRecord`.

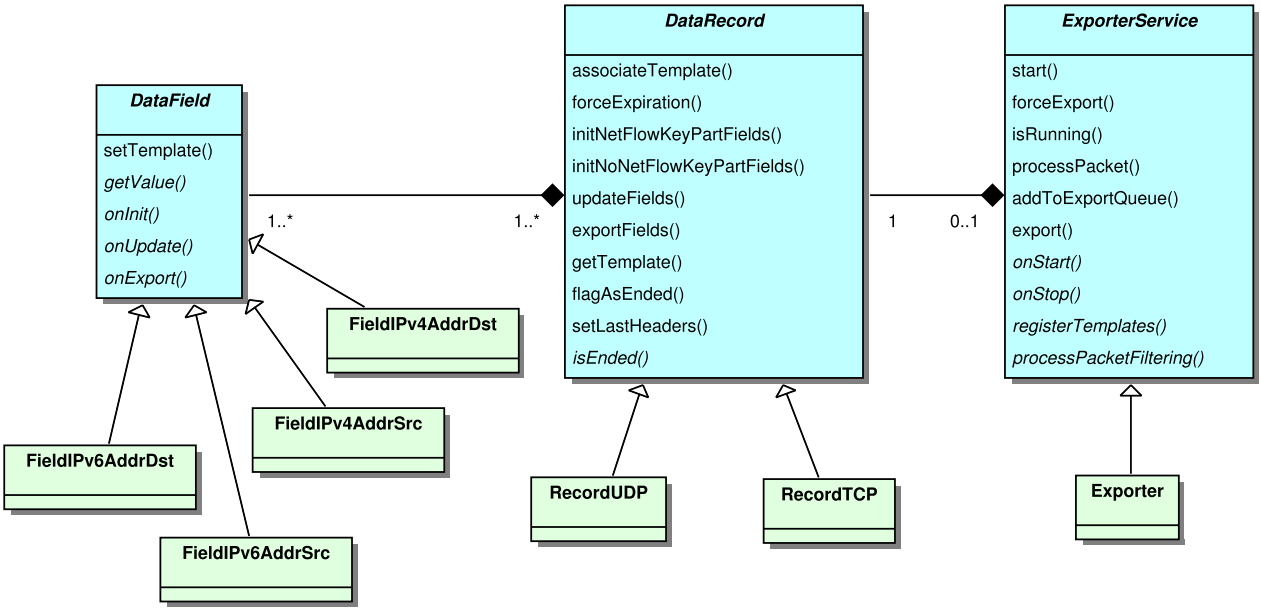

FIGURE 1.4 – Diagramme UML d'exemple d'implémentation de la bibliothèque.

Enfin, un `DataFlowSet` devra être créé pour accueillir une série de `DataRecord`. L'implémentation de la bibliothèque vérifiera que tous les `DataRecord` ajoutés au `DataFlowSet` se réfèrent tous au même `TemplateRecord`, dont il reprendra l'identifiant, comme spécifié par la RFC.

Diverses vérifications sont faites (comme les intervalles d'identifiants autorisés) par la bibliothèque au moment des associations pour s'assurer que le paquet d'export final correspond bien aux directives du standard.

La section suivante propose un exemple simple et concret d'implémentation, permettant de voir plus en détail les différentes possibilités proposées par la bibliothèque.

1.3.5 Implémentation de la bibliothèque

1.3.5.1 Conception et prérequis

La figure 1.4 propose un exemple simplifié d'implémentation de la bibliothèque (les méthodes en italique dans les classes abstraites de la bibliothèque — en bleu — sont obligatoirement implémentées dans les classes sous-jacentes).

Nous retrouvons les différentes classes qui héritent de `DataField` et qui indiquent à l'exportateur comment extraire l'information qu'elles représentent. Deux classes héritent de `DataRecord`, pour représenter des NetFlows correspondant à des flux TCP comme UDP. Les modèles ne sont pas représentés parce qu'ils n'impliquent pas la création de nouvelles classes.

Les prérequis pour implémenter la bibliothèque sont les suivants :

1. le SDK (*Software Development Kit*) de Android utilisé doit au minimum être en version neuf ;
2. le téléphone doit être *rooté* (accès au super-utilisateur) ;
3. la bibliothèque (*madynes-owoid-exporter.jar*) doit être ajoutée au chemin de compilation java du projet ;
4. en cas d'utilisation du pilote pour IP, il est également conseillé d'ajouter la bibliothèque du parseur (*netutils-parse-v1.jar*) ;
5. les binaires des pilotes doivent être copiés dans le répertoire */res/raw/* du projet ;
6. le fichier de déclaration Android utilisé doit au minimum comporter les permissions suivantes :

```
<uses-permission android:name="android.permission.ACCESS_SUPERUSER" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
```

La prochaine étape est de créer la classe correspondant à l'exportateur personnalisé.

1.3.5.2 Création de l'exportateur

Il suffit de créer une classe qui hérite de `ExportService` (*Exporter.java*) :

```
import madynes.flowoid.exporter.*;
import madynes.flowoid.exporter.TemplateRecord.*;
import madynes.flowoid.exporter.TemplateField.InadequateLengthException;
import madynes.flowoid.exporter.TemplateFlowSet.TemplateNotFoundException;

public class Exporter extends ExporterService {

    @Override
    public void registerTemplates() throws AlreadyUsedIDException,
        InadequateIDException, InadequateLengthException {

    }

    @Override
    public void processPacketFiltering(byte[] headers) throws
        TemplateNotFoundException {

    }

    @Override
    public void onStart() {

    }

    @Override
    public void onStop() {

    }
}
```

Les fonctions `onStart()` et `onStop()` sont appelées respectivement au démarrage et à l'arrêt de l'export des NetFlows. Les autres fonctions sont décrites dans les sections suivantes.

1.3.5.3 Création des modèles

Touts les modèles doivent être déclarés dans la fonction `registerTemplates()` de l'exportateur.

1. La première étape est de créer tous les modèles de champ en instanciant des objets `TemplateField`. Le constructeur nécessite un identifiant parmi ceux disponibles dans `TemplateField .TypeDefinition.*`. Si la taille par défaut est zéro, cela signifie qu'elle est variable et qu'il faut donc utiliser l'autre constructeur qui prend en second argument la taille désirée en octets.

   Exemple :
   `TemplateField portSrc = new TemplateField(TemplateField.TypeDefinition.L4_SRC_PORT);`

2. Il faut ensuite créer les modèles de NetFlow en instantiant des objets `TemplateRecord` et en choisissant un identifiant entre 256 et 65535 (sous peine sinon de lever une exception). Cet identifiant ne peut pas être généré automatiquement puisque le collecteur doit aussi en avoir connaissance pour appréhender la sémantique des NetFlows associés.

Exemple :
`TemplateRecord ipv6TcpTemplate = new TemplateRecord(257);`

3. Les modèles de champ doivent être ensuite associés aux modèles de NetFlows à l'aide de la fonction `addTemplateField(TemplateField)`, en gardant à l'esprit que l'ordre d'ajout est significatif. Un même `TemplateField` peut être associé à plusieurs `TemplateRecord` différents.

   Exemple :
   `ipv6TcpTemplate.addTemplateField(portSrc);`

4. La dernière étape est d'enregistrer le modèle de NetFlow avec la fonction `registerTemplate Record(RecordTemplate)` pour qu'il soit utilisable par la suite.

   Exemple :
   `registerTemplateRecord(ipv6TcpTemplate);`

1.3.5.4 Remplissage des NetFlows

Les modèles de champ déterminent comment une valeur est représentée et — pour ceux qui sont standardisés — ce qu'elles représentent. Les champs de données déterminent comment la valeur doit être récupérée sur le système. Par conséquent chaque modèle de champ doit avoir un champ de données correspondant.

Pour créer un champ de données, il faut ajouter une classe qui étend la classe `DataField`. Puis implémenter les fonctions abstraites suivantes :

1. `onInit(byte[])` : appelée automatiquement chaque fois qu'un nouveau NetFlow est initié. Son unique argument correspond au premier paquet reçu pour le NetFlow. Dans notre exemple, c'est donc cette méthode qui sert à extraire l'adresse IP source qui sera commune à tous les paquets qui suivront pour le même NetFlow.

2. `onUpdate(byte[])` : appelée automatiquement chaque fois qu'un nouveau paquet est capturé pour un NetFlow déjà existant. Pour un champ correspondant au nombre de paquets du flux, c'est cette méthode qui sert pour incrémenter un simple compteur.

3. `onExport(byte[])` : appelée automatiquement lorsque la dernier paquet du NetFlow est reçu, juste avant que le paquet d'export qui le contient ne l'envoie au collecteur. Plus précisément, il s'agit du paquet qui marque la fin du flux (e.g. pour lequel la méthode `isEnded()` du `DataRecord` que nous verrons juste après renvoie vrai), mais pas obligatoirement le tout dernier paquet. Par exemple pour un flux TCP, ce sera le paquet contenant le drapeau `FIN` ou `RST` mais pas celui du `ACK` final. Pour un flux UDP par contre, ce sera le tout dernier paquet reçu, juste avant la date d'expiration du flux.

4. `getValue()` : appelée automatiquement quand l'exportateur crée le paquet d'export. Son implémentation doit utiliser la fonction héritée `getByteBuffer()` qui renvoie un tampon d'octets de la taille imposée par le modèle associé. Par exemple pour un modèle indiquant une taille de quatre, il faudra utiliser la méthode `putInt(int)` pour remplir le tampon avec la valeur récupérée au cours de la constitution du NetFlow, via l'une des trois méthodes décrites ci-dessus.

Exemple d'implémentation (*FieldTCPPortSrc.java*) :

```
1 import java.nio.ByteBuffer;
2 import edu.huji.cs.netutils.parse.TCPPacket;
3
```

```
4  public class FieldTCPPortSrc extends DataField {
5    private short value;
6
7    @Override
8    public void onInit(byte[] headers) {
9      TCPPacket tcpPacket = new TCPPacket(headers);
10     value = (short) tcpPacket.getSourcePort();
11   }
12
13   @Override
14   public void onUpdate(byte[] headers) {}
15
16   @Override
17   public void onExport(byte[] headers) {}
18
19   @Override
20   public ByteBuffer getValue() {
21     ByteBuffer bytes = getByteBuffer();
22
23     bytes.putShort(value);
24
25     return bytes;
26   }
27 }
```

1.3.5.5 Marqueur de fin des NetFlows

Un nouveau NetFlow (donc un objet héritant de `DataRecord`) est créé dès que la combinaison de ses champs discriminants n'est pas encore connue, ou a déjà été exportée. Ce point est précisé dans les sections qui suivent.

Les conditions pour décider qu'un NetFlow doit être arrêté sont vérifiées à chaque nouveau paquet qui correspond au NetFlow. Si ces conditions sont validées, le `DataRecord` est déplacé de la liste des NetFlows en cours vers la file d'exportation. Puisque ces conditions sont spécifiques au type de flux, pour chaque `TemplateRecord`, il faut écrire une classe qui étend `DataRecord` et qui implémente la fonction abstraite `boolean isEnded(byte[])`. L'argument correspond aux entêtes du nouveau paquet ajouté au NetFlow, qui servira éventuellement à détecter la fin d'un flux.

Par exemple (*RecordTCP.java*) :

```
1  import madynes.flowoid.exporter.TemplateRecord;
2  import edu.huji.cs.netutils.parse.TCPPacket;
3
4  public class RecordTCP extends DataRecord {
5
6      public RecordTCP(TemplateRecord template) {
7          super(template);
8      }
9
10     @Override
11     public boolean isEnded(byte[] headers) {
12         TCPPacket tcpPacket = new TCPPacket(headers);
13
14         return tcpPacket.isFin() || tcpPacket.isRst();
15     }
16
17 }
```

Dès lors que cette fonction renverra vrai, le NetFlow sera marqué comme fini. Comme expliqué dans la section précédente pour la fonction `onExport(byte[])`, un temps d'expiration suivra pour récolter les éventuels paquets qui suivraient le paquet qui annonce la fin du flux (ce sans quoi les flux TCP seraient systématiquement coupés en deux, avec un second flux d'un seul paquet d'acquittement).

Pour tous les types de flux, l'exportateur dispose de temps d'expiration, qui lui permet de prendre la décision d'exporter le NetFlow, coupant ainsi le flux en plusieurs NetFlows. Cette mesure est prise pour les connexions qui seraient trop longues et qui remonteraient donc trop tard au collecteur, ou prendraient trop de place en mémoire. Pour les flux qui ne disposent pas de marqueur de fin, comme avec UDP, ce temps d'expiration est indispensable.

La file d'attente d'exportation n'est vidée que lorsque le nombre minimum de NetFlows est atteint, ou que le temps d'expiration du paquet d'exportation en attente de départ est dépassé (permettant aux NetFlows qui se trouveraient sur un périphérique qui a très peu de communications de malgré tout rejoindre le collecteur régulièrement).

1.3.5.6 Associer les modèles aux données

Le lien entre les objets `DataRecord` (qui sont chargés de récupérer la valeur des champs) et leur modèle doit être assuré en tête de l'implémentation de la fonction `processPacketFiltering(byte[])`.

1. La première étape consiste à récupérer le modèle à associer, qui a été créé lors de l'appel unique de la fonction `registerTemplates()`. L'identifiant du modèle doit être utilisé pour le désigner (il est conseillé de passer par une constante de l'exportateur, par exemple `TPL_IPV4_TCP_ID`, pour l'enregistrement et la récupération du modèle).

   Exemple :
   ```
   TemplateRecord template = getRegistredTemplateRecord(257);
   ```

2. Les objets `DataRecord` doivent être instanciés à partir des classes précédemment crées, en leur associant l'un des modèles qui vient d'être récupéré.

   Exemple :
   ```
   record = new RecordTCP(template);
   ```

3. L'étape suivante consiste à associer chacun des `TemplateField` du `TemplateRecord` avec un `DataField`. Ils doivent être ajoutés au `DataRecord` via la fonction `associateTemplateWithData (DataRecord, boolean)` dans le même ordre que les modèles ont été ajoutés au `TemplateRecord` associé (sous peine sinon de lever une exception). Le second argument permet de préciser si le champ sera discrimant ou non dans l'identité du NetFlow (voir la section suivante).

   Exemple :
   ```
   record.associateTemplateWithData(new FieldIPv6AddrSrc(), true);
   ```

4. Enfin, le traitement du paquet par l'exportateur doit être demandé en utilisant la fonction `processPacket(DataRecord, byte[])` héritée de `ExporterService`.

   Exemple :
   ```
   processPacket(record, headers);
   ```

Le `DataRecord` (et plus précisément les champs déclarés comme étant discriminants) sera utilisé pour déterminer si le paquet appartient à un NetFlow existant ou non (traitement `onInit(byte[])`). Si c'est le cas, le paquet sera ajouté à l'instance en cours (traitement `onUpdate(byte[])`), sinon le `DataRecord` sera ajouté aux NetFlows en cours d'observation.

1.3.5.7 Filtrage des paquets

Tous les entêtes de paquets capturés par les pilotes sont envoyés à la fonction `processPacket Filtering(byte[])`. Pour créer les `DataRecord` qui correspondent au type de paquet, il faut donc enrichir cette fonction de conditions de filtrage et de vérification des entêtes.

Par exemple :

```
1  @Override
2  public boolean processPacketFiltering(byte[] headers) throws
       TemplateNotFoundException {
3      LocationFinder locationFinder = LocationFinder.getInstance(this);
4      DataRecord record = null;
5
6      try {
7          // TCP / IPv6
8          if(EthernetFrame.statIsIpv6Packet(headers)
9              && IPFactory.isTCPPacket(headers)) {
10
11             TemplateRecord template = getRegistredTemplateRecord("ipv6-tcp");
12             record = new RecordTCP(template);
13
14             // Champs discriminants
15             record.associateTemplateWithData(new FieldIPv6AddrSrc(), true);
16             record.associateTemplateWithData(new FieldIPv6AddrDst(), true);
17             record.associateTemplateWithData(new FieldTCPPortSrc(), true);
18             record.associateTemplateWithData(new FieldTCPPortDst(), true);
19
20             // Autres champs
21             record.associateTemplateWithData(new FieldIPv6Length(), false);
22         }
23
24         // Autres types de paquet
25         else if(...) {
26             ...
27         }
28
29         // Traitement (ou non) du paquet
30         if(record != null) {
31             processPacket(record, headers);
32         }
33
34     // Erreur du parseur
35     } catch(...) {
36         return false;
37     }
38
39     return true;
40 }
```

1.3.5.8 Ajouter un pilote

Les pilotes sont écrits en C pour accéder aux couches les plus basses du système et ainsi pouvoir écouter directement les interfaces réseaux.

La seule prérogative est d'envoyer les entêtes de chacun des paquets écoutés à l'exportateur java à travers une *socket* TCP interne. Le pilote doit également accepter en arguments le nom d'une interface suivie d'une liste de filtres. Du côté de l'exportateur java il faut simplement créer les objets décrits plus haut avec un parseur adéquat pour filtrer les paquets et extraire les champs intéressants.

Dans le cas du pilote IP, le plus gros problème a résidé dans la différence de traitement entre les réseaux wifi et la 3G sur les versions récentes de Android. Dans ce second cas, une fausse interface réseau est utilisée avec un tunnel. Lorsque la librairie C *libpcap* utilisée pour capturer le réseau rencontre ce type de trafic, elle ne peut pas interpréter la couche liaison de données, et décide donc de la remplacer par son propre format *cooked*. En envoyant les entêtes telles quelles au parseur Java, ce dernier ne pouvait pas les exploiter. Le pilote IP reconstruit donc parfois de toutes pièces une fausse couche liaison de données ethernet à partir du format *cooked* pour que le parseur puisse faire son travail, y compris avec les paquets du réseau 3G.

Le parseur Java utilisé pour le pilote IP dans Flowoid v2 est la seule partie qui n'a pas été écrite

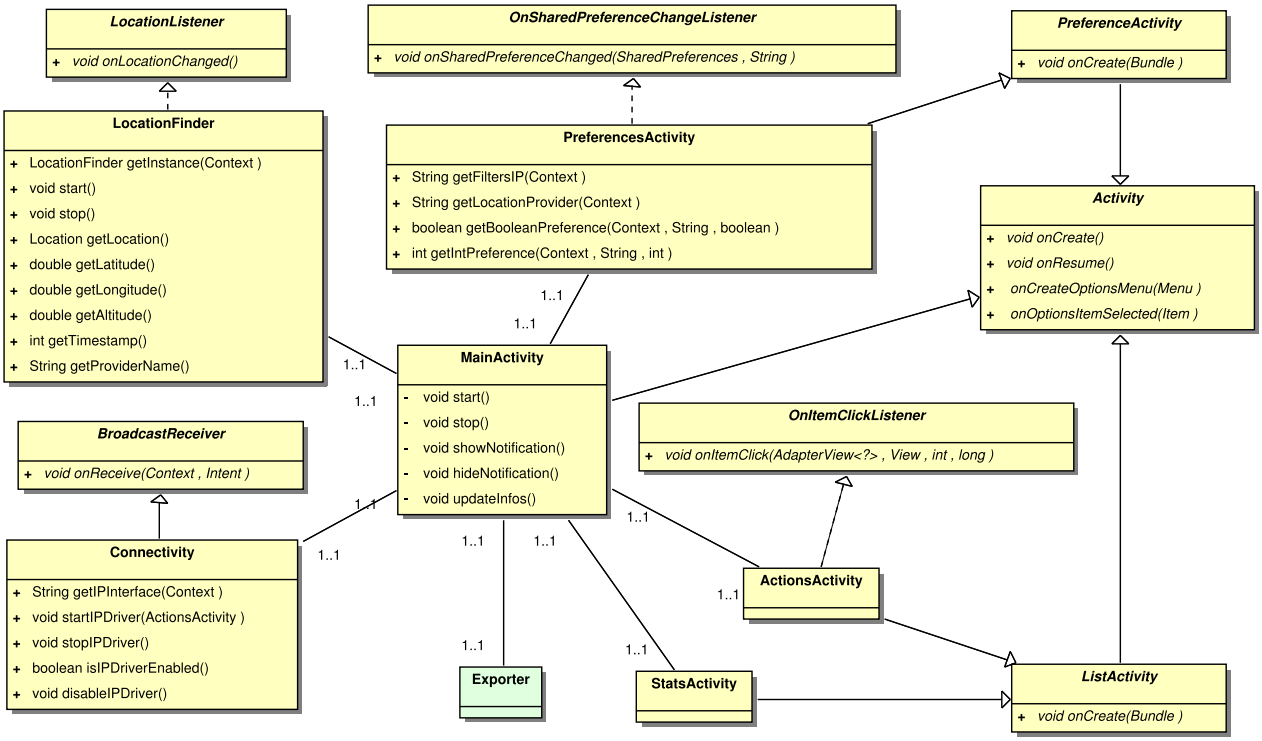

FIGURE 1.5 – Diagramme UML de Flowoid v2.

spécifiquement pour le projet. Il s'agit d'une partie spécifique de la bibliothèque Netutils [2], qui a été recompressée seule et ajoutée à Flowoid v2. Elle est délivrée en licence libre GPL v2, et a pour seul défaut de ne pas supporter complétement IPv6 (notamment les extensions, même si elles ne sont pas beaucoup utilisées dans un contexte mobile).

Pour ajouter un pilote, il suffit de placer les sources dans le répertoire *jni/drivers/* de la bibliothèque et d'utiliser le *Makefile* (en adaptant éventuellement les options) fourni pour le construire. Le binaire final doit être copié dans le dossier *res/raw/* du projet de l'implémentation. Juste après l'appel de démarrage de l'exportateur, il faut utiliser `Driver.useDriver` avec le nom du pilote, le nom de l'interface sur laquelle il doit écouter ainsi que les filtres à utiliser.

Par exemple pour le pilote IP fourni :
```
Driver.useDriver(this, R.raw.driver_ip, "eth0", "ip6 and tcp");
```

Il suffit enfin de réinstaller l'application Android.

1.3.6 Utilisation de l'exportateur

1.3.6.1 Conception

Flowoid v2 est un exemple d'implémentation de la bibliothèque présentée précédemment. Il est destiné à exporter les NetFlows créés depuis l'activité réseau d'un périphérique Android, tout en proposant une interface graphique aux utilisateurs. La figure 1.5 présente une version simplifiée du diagramme UML des classes et méthodes publiques utilisées, en s'appuyant sur une implémentation de la bibliothèque (en vert).

La première version du projet Flowoid a été écrite par l'équipe Madynes avant mon arrivée. Il reposait sur le logiciel SoftFlowd pour l'export au format NetFlow et ajoutait quelques options d'export via son interface graphique. Chaque modification était lourde à implémenter et l'exportateur était soumis à de multiples licences. Avec la bibliothèque écrite pour l'occasion, Flowoid v2 est une réécriture complète des deux projets.

Les sections suivantes présentent les principales classes du projet.

2. https://code.google.com/p/netutils/

1.3.6.2 Intégration de la géolocalisation

Flowoid est, entre autres, destiné à exploiter l'exportateur pour associer des coordonnées géographiques aux flux observés sur l'équipement. Celles-ci sont dont récupérées pour chaque NetFlow transmis. Pour ce faire, un service tourne en permanence sur le système Android, qui le prévient dès que les coordonnées du périphérique sont suffisamment modifiées.

L'utilisateur peut entièrement paramétrer la méthode de localisation, en utilisant :

- la méthode passive :
- le réseau ;
- le GPS.

Dans le cas de la méthode passive (par défaut), l'application consulte la dernière localisation récupérées par le périphérique via une autre application. C'est l'option la moins énergivore mais également la moins précise. La méthode par le réseau est un peu plus fiable mais implique une précision qui va souvent jusqu'à la centaine de mètres. La méthode par le GPS est la plus précise, mais elle est extrêmement énergivore pour l'utilisateur, qui a peu de chance de souhaiter l'activer. Elle a aussi l'inconvénient de ne pas fonctionner en intérieur.

La classe `LocationFinder` s'occupe de cette tâche, en se positionnant en écoute dès le lancement du programme en réclamant le niveau de précision demandé par l'utilisateur. Dans le cas d'une écoute passive, elle préfère lancer un chronomètre qui ira régulièrement vérifier la dernière valeur enregistrée. Différents tests ont prouvé que cette façon de faire, particulièrement bien adaptée pour la méthode passive, était beaucoup moins énergivore que l'écoute permanente des changements de localisation. À tout instant, il est possible de la consulter de manière statique pour connaître les dernières coordonnées récupérées.

Lors de l'étape de l'implémentation de la bibliothèque, des classes comme `FieldLocLatInt`, `FieldLocLatDec`, `FieldLocLongInt` et `FieldLocLongDec` ont été crées pour récupérer la valeur de `LocationFinder` lors de l'initialisation des NetFlows. La séparation en entiers et décimales a été souhaitée par notre correspondant de l'université de Twente, qui développe le collecteur qui reçoit les données et se focalise sur leur exploitation. La localisation des NetFlows sera utilisée pour la partie recherche du stage, afin de pouvoir déterminer la localisation de l'utilisateur dans le but de comparer les valeurs aux résultats du modèle de recherche.

1.3.6.3 Connectivité réseau

Le pilote IP qui écoute les paquets sur les interfaces réseau a besoin de savoir quelle interface écouter. Par exemple si l'utilisateur active le wifi, il faut cesser d'écouter sur l'interface correspondant au réseau 3G pour écouter l'interface correspondant à l'antenne wifi.

Ainsi, il faut être capable de déterminer l'interface active au lancement du pilote et détecter les changements liés au réseau. Pour ce faire, Flowoid v2 utilise sa classe `Connectivity` qui demande au système Android de le prévenir dès qu'il y a un changement. Plusieurs méthodes ont été testées pour déterminer l'interface du réseau IP active, et la seule qui ait fait ses preuves quelque soit le type de périphérique ou la version du système, est de regarder directement dans la table de routage. Ainsi, en consultant les fichiers du système on peut trouver l'interface qui est utilisée comme route par défaut, et qui est donc probablement celle qui nous apportera le plus d'informations.

Cette recherche est refaite chaque fois que le système indique un changement d'état du réseau. Elle implique l'arrêt du pilote IP et de le relancer avec la nouvelle interface détectée, de façon entièrement automatique.

1.3.6.4 NetFlows

La puissance et la modularité de la bibliothèque de l'exportateur a pleinement été exploitée dans Flowoid v2.

Ainsi, pas moins de ving-six classes `DataField` et quatre classes `DataRecord` ont été créées pour former les six types de NetFlows possibles (avec autant d'instances de `TemplateRecord` créées) :

- modèles pour IPv6 (TCP, UDP et ICMPv6) ;
- modèles pour IPv4 (TCP, UDP et ICMP).

Les modèles se différent par le type d'adresse (IPv6 ou IPv4) et les informations disponibles selon le type de protocole de transport (i.e. les ports pour TCP et UDP). Un lot important de champs sont partagés par tous les modèles (localisation, taille du flux, nombre de paquets, dates, nom de l'application, etc.).

1.3.6.5 Nom des applications

Un champ de données particulier a dû être ajouté aux NetFlows gérés par Flowoid v2. Il s'agit du nom de l'application associée au flux.

Celui-ci a particulièrement été compliqué à associer. La seule solution viable trouvée a été de consulter directement le fichier système qui rassemble l'ensemble des connexions sur le système, en recherchant la ligne concernée à l'aide du numéro de port applicatif source associé. Cette solution permet d'extraire l'identifiant de l'utilisateur système qui est associé à la connexion.

Ce champ peut contenir plusieurs types de valeur, dans cet ordre de préférence :

1. le système Android utilise une série d'utilisateurs systèmes pour ses différentes routines natives. Cette liste a été intégrée à Flowoid v2 pour permettre au champ de données d'être complété avec le nom de la routine qui correspond ;

2. si l'identifiant ne correspond à aucun des identifiants connus, c'est l'API de Android qui est utilisée. La machine virtuelle Dalvik utilise automatiquement un utilisateur système par application et l'API permet d'associer les deux, permettant au champ de données d'être rempli de façon précise ;

3. si l'identifiant n'a pas pu être associé à un nom d'application, l'identifiant numérique est directement utilisé en le préfixant de la chaîne `uid:` (par exemple, un binaire directement lancé dans un interpréteur Bash pour obtenir les droits de super-utilisateur donnera toujours `uid:0`) ;

4. enfin, il existe des cas où aucune ligne correspondant au port recherché n'est trouvée, auquel cas la chaîne `Unknown` est utilisée (cette réponse est utilisée pour quasiment chacune des requêtes DNS, qui sont suffisamment courtes pour terminer avant que l'application n'ait le temps de consulter le fichier système).

1.3.6.6 Interface utilisateur

Outre les principales fonctionnalités décrites ci-dessus qui permettent de donner les informations adéquates aux pilotes et aux différents `DataField` implémentés pour l'exportateur, Flowoid v2 a surtout pour but de présenter une interface un minimum conviviale et paramétrable pour les utilisateurs. La figure 1.6 présente l'écran d'accueil de Flowoid v2 au moment de l'activation de l'exportateur.

De haut en bas, on peut y voir :

– la petite icône de Flowoid (reprise de la première version) qui apparait dans la barre des tâches et qui restera visible sur le périphérique tant que l'exportateur sera actif ;

– le gros bouton avec la barre verte qui permet de démarrer ou arrêter l'exportateur d'un coup de doigt (grisé sur la capture d'écran parce que l'exportateur est en train de s'activer) ;

– la première ligne « *Connected Drivers* » qui indique le nombre de pilotes connectés à l'exportateur, et qui doit au minimum être à `1` pour que l'exportateur ait une utilité (en rouge sur la capture d'écran parce que le pilote IP n'a pas encore rejoint l'exportateur) ;

– la seconde ligne « *Active Netflows* » qui indique en temps réel le nombre de NetFlows considérés actifs (et donc le nombre de `DataRecord` instanciés) ;

– la ligne « *Export Queue* » qui indique le nombre de NetFlows considérés comme terminés ou expirés et qui attendent le prochain départ vers le collecteur à bord d'un paquet d'export ;

– la ligne « *IP Network Interface* » qui indique l'interface réseau jugée active qui a été trouvée par l'application (et qui peut porter des noms très différents d'un système à l'autre) ;

– la ligne « *Location Provider* » qui indique la méthode choisie par l'utilisateur pour déterminer la localisation géographique du périphérique, avec un lien vers une carte pointant exactement sur l'endroit correspondant au dernier lieu enregistré ;

– la ligne « *Device ID* » légèrement recouverte qui permet de connaître l'identifiant qui sera utilisé pour identifier le périphérique au niveau du collecteur ;

– le message qui recouvre la ligne précédente qui s'affiche durant une seconde pour indiquer à l'utilisateur que l'exportateur est bien en train de s'activer ;

– et enfin la barre du bas qui se remplit au fur et à mesure que le nombre de NetFlows en mémoire grandit (avec une limite paramétrable par l'utilisateur, qui arrête la création de nouveaux `DataRecord` dès lors qu'elle est atteinte et qui la reprend dès lors qu'un paquet d'export a libéré de la place).

Concernant les options de l'application, c'est naturellement le menu traditionnel d'Android qui a été utilisé. Comme on peut le voir sur la figure 1.7, celui-ci propose quatre choix.

Un exemple de sous-menu est disponible en figure 1.8. Il s'agit du menu dédié au seul pilote actuellement disponible. Il permet de l'activer ou non, de choisir la ou les versions du protocole IP à prendre en compte et de préciser les protocoles de transport à écouter (sur cette capture d'écran ICMP a été désactivé automatiquement à la désactivation de IPv4 puisqu'il n'est pas utilisé par IPv6).

Le menu préférences propose pas moins de vingt-cinq paramètres personnalisables organisés en sous-menus, que nous nous interdiront de lister un à un dans ce rapport pour en préserver sa légèreté.

Un écran dédié aux statistiques de l'exportateur est également disponible (visible en figure 1.9), proposant un large de choix de métriques permettant de vérifier l'activité de l'application. Cette fonctionnalité est gérée par la classe `StatsActivity`.

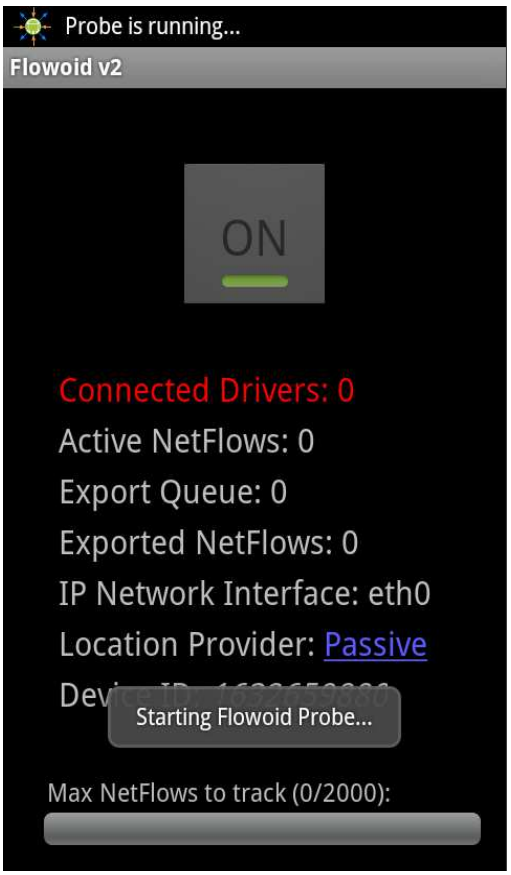

FIGURE 1.6 – Capture d'écran de Flowoid v2 (lancement).

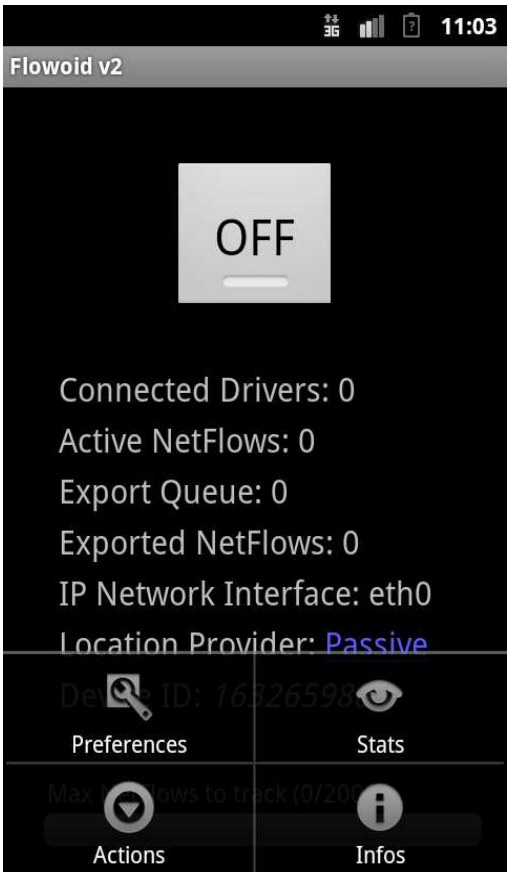

FIGURE 1.7 – Capture d'écran de Flowoid v2 (menu options).

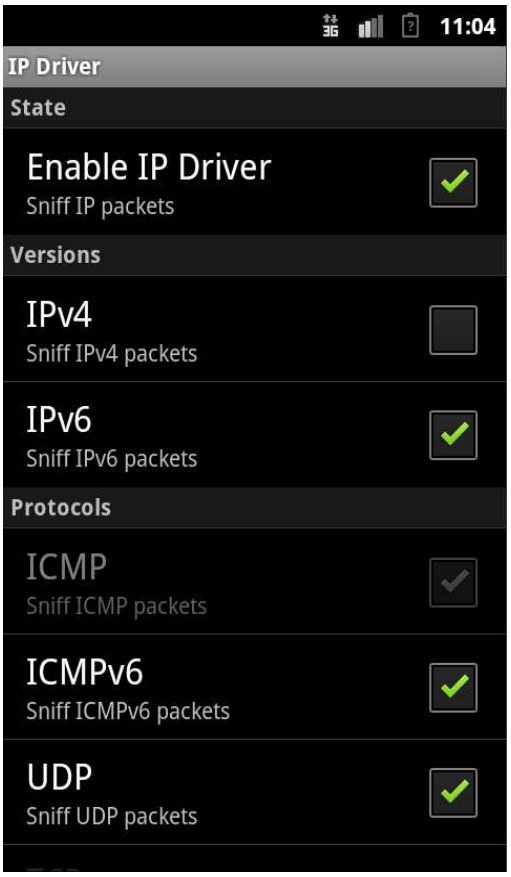

FIGURE 1.8 – Capture d'écran de Flowoid v2 (paramétrage du pilote IP).

FIGURE 1.9 – Capture d'écran de Flowoid v2 (statistiques).

Chapitre 2

Inférence de la localisation

Sommaire

Cette section présente la partie recherche du stage, qui a consisté en l'élaboration d'un modèle destiné à vérifier s'il est possible de trouver des patrons qui permettent d'inférer la fréquence de la visite d'un type de lieu (sans déterminer sa nature) où l'utilisateur se situe.

2.1 Introduction

Internet est une grande suite de routeurs situés partout dans le monde et derrière chacun d'entre eux se cache un fournisseur d'accès à Internet (FAI). Le type de FAI le plus connu est le FAI de bordure qui permet aux utilisateurs finaux de se relier au réseau mondial. Ceux qui permettent aux utilisateurs d'atteindre l'autre bout du monde sont les FAI (*ISP*) qui servent de transitaires, et qui se situent au milieu du réseau.

Alors que les opérateurs de réseau ne devraient être que des vendeurs de tuyaux, les gouvernements du monde entier les pointent de plus en plus dans le but de leur donner la charge de surveiller ce qu'ils transportent, pour les faire devenir les parfaits gardiens de la bonne morale sur Internet. Les récents exemple sont les lois SOPA [1], PIPA [2] et ACTA [3] aux USA, CETA [4] au Canada et dans l'Union Européenne ou encore Hadopi [5] et LOPPSI [6] en France. Toutes ces lois sont supposées combattre les activités illégales qui utilisent Internet, comme la pédophilie et les violations de doits d'auteur (en particulier dans l'industrie musicale et cinématographique). La principale conséquence

1. http://news.cnet.com/8301-31921_3-57329001-281/how-sopa-would-affect-you-faq/
2. http://www.cbsnews.com/8301-503544_162-57360665-503544/sopa-pipa-what-you-need-to-know/
3. http://publicknowledge.org/issues/acta
4. http://www.pcworld.com/article/259092/no_isp_police_rule_in_canadaeu_trade_agreement_says_spokesman.html
5. http://www.laquadrature.net/fr/hadopi
6. http://www.regardscitoyens.org/category/loppsi/

pour les FAI de bordure est une pression de plus en plus forte pour aller impiéter sur la vie privée de leurs utilisateurs pour regarder ce qu'ils font de leur accès à Internet [7].

Dans un même temps, l'utilisation d'Internet est en train d'évoluer et de plus en plus de gens accèdent au réseau hors de chez eux, lors de leurs déplacements (dans la rue, dans les bars, durant leurs vacances, etc.). Ils utilisent des tablettes ou des ordiphones (les ventes de tablettes dépasseront celles des petits ordinateurs durant l'année 2013 [8]) pour se connecter à leur FAI partout où ils vont. Beaucoup des applications de ce type d'appareil sont autorisées à envoyer la position géographique du périphérique sur Internet en permanence, sans demande d'autorisation. Heureusement, quelques utilisateurs s'inquiètent de leur vie privée et font attention à n'installer qu'une poignée d'applications qui ont l'autorisation de récupérer leur position géographique, ou désactivent directement la possibilité de les tracer dans les paramètres de leur périphérique. Leur FAI est alors le seul qu'ils ne peuvent pas contrôler parce que lui seul peut voir leur trafic tout le temps et n'importe où.

Grâce aux technologies comme le DPI (*Deep Packet Inspection*) [1], les FAI sont capables de voir des informations très fines comme les adresses des sites web consultés ou le contenu des pages, dans le cas des sessions non-chiffrées (à l'exception des FAI très intrusifs qui interceptent et remplacent les certificats mais ce sont des exceptions très spécifiques). Pour les autres types de sessions (e.g. utilisant du chiffrement comme le SSL ou n'étant pas examinées par du DPI) il ne peuvent voir que les adresses IP, les ports applicatifs, la taille des données et s'il y a un trafic ou non à un moment donné. Dans la mesure où le DPI est une technologie très coûteuse, les FAI sont quasiment toujours dans le second cas, avec uniquement les informations des couches trois et (potentiellement) quatre d'accessibles. L'utilisation de IPsec [RFC 2401] serait une bonne solution pour restreindre encore plus les information disponibles, en limitant uniquement aux informations de la couche trois (seulement les adresses IP) mais cette technologie n'est quasiment jamais utilisable pour les utilisateurs finaux. En supposant que les VPN (*Virtual Private Networks*) sont une bonne solution mais qu'ils ne sont pas accessibles pour la majorité des internautes, les informations standards qu'on peut trouver dans des NetFlows sont les informations qu'un FAI peut généralement trouver au niveau d'un FAI qui ne va pas plus loin que les ports applicatifs de la couche transport.

Depuis 1996 (dépôt du brevet sur les NetFlows par Cisco), beaucoup de papiers de recherche ont fait état de l'utilisation des NetFlows pour résoudre des problèmes de sécurité ou de vie privée. La sécurité est le sujet le plus largement traité avec des problématiques comme la détection d'intrusion [3], la détection du *spamming* [4] ou encore la détection des réseaux de robots (*botnets*) [5] et d'autres [6]. Côté vie privée, d'après le travail de Melnikov et al. [7], un utilisateur peut être caractérisé par son activité sur Internet en analysant ses enregistrements de NetFlows, permettant probablement aux FAI de tracer les utilisateurs non-identifiés au gré de leurs changements d'ordinateurs ou de connexion à Internet. Le prochain sujet pourrait être de cibler le type d'activité de l'utilisateur. Ainsi, Soysal et al. [9] ont étudiés plusieurs méthodes pour inférer le type de trafic (et donc le type d'application utilisé), évitant l'utilisation du DPI qu'ils considèrent eux-mêmes comme une « *violation de la vie privée des utilisateurs* » (*violation of user data privacy*). Puisque qu'ils considèrent que l'inférence par l'utilisation des ports applicatifs n'est pas efficace, leur travail est basé sur une méthode de *machine learning* qui utilise une classification sur la base des flux réseaux. Nous aurons l'opportunité plus tard de confirmer que la méthode basée sur les ports applicatifs n'est pas efficace, en particulier dans un contexte mobile.

Les études autour de la vie privée utilisant les NetFlows semblent ne pas beaucoup considérer les nouvelles possibilités offertes par le nouvel usage mobile du réseau Internet. Si nous supposons que nous sommes capables d'identifier les utilisateurs et le type d'application qu'ils utilisent à partir de leurs NetFlows, nous pourrions étudier si nous sommes aussi capables de profiler leur localisation avec ces mêmes informations, même si l'utilisateur a interdit l'usage de la détection de sa position géographique.

La section suivante analyse les informations communément accessibles depuis des enregistrements NetFlows et leur pertinence vis-à-vis de notre problématique.

7. http://www.guardian.co.uk/world/2013/jun/07/prism-tech-giants-shock-nsa-data-mining
8. http://www.eetimes.com/electronics-news/4404850/Tablet-sales-to-outpace-notebooks

Les NetFlows transportent en général six champs intéressants :

- adresses IP source/destination ;
- ports applicatifs source/destination ;
- date et taille.

2.2 Caractérisation des flux réseaux

2.2.1 Ports applicatifs

Dans la plupart des cas, un seul port par NetFlow peut être interprété étant donnée l'utilisation des ports dynamique insignifiants qu'utilisent la plupart des protocoles de transport. Le seul port significatif (de destination dans le cas d'un flux sortant) est supposé décrire l'application qui est utilisée. Mais dans les faits, et après avoir analysé des jeux de données d'utilisateurs, les seuls ports qui sont réellement significatifs pour un usage standard d'ordiphone sont :

1. la messagerie instantanée ;
2. les courriels.

La plupart des autres applications utilisent quasiment exclusivement les ports 80 et 443, qu'on retrouve pour de la visite de sites web, de la mise à jour d'applications, de la récupération de fichiers distants, de la synchronisation de données, etc.

2.2.2 Adresses IP source

L'adresse IP source (dans le cas d'un flux sortant) pourrait surtout nous donner des informations à propos de l'utilisation ou non du wifi. Ainsi, si l'adresse IPv4 fait partie des blocs d'adresses privés de la RFC 1918, à l'exception du bloc 10/8, ou est de type publique, alors l'utilisateur n'utilise probablement pas une connexion 3G mais une connexion wifi. L'inverse n'étant pas vrai, puisqu'une adresse dans le bloc 10/8 pourrait aussi provenir d'un réseau wifi.

Dans le cas de l'IPv6, on pourrait malheureusement conclure quasiment systématiquement qu'il s'agit d'une connexion wifi, étant donné le retard dramatique des opérateurs mobiles dans ce domaine.

2.2.3 Adresses IP destination

L'adresse IP de destination est l'information la plus intéressante à notre disposition. Mais si nous voulions déduire une information depuis un flux en se basant sur son adresse IP de destination, nous aurions à utiliser une catégorisation de toutes les adresses IP du monde par rapport au type de service qu'elles permettent de consulter. Ce type de classification existe pour les destinations dangereuses ou malveillantes (i.e. listes de l'université de Toulouse) mais jamais pour toutes les IP du réseau Internet. De plus une adresse IP seule ne peut pas décrire le type de service à l'autre bout de la chaîne, à cause des serveurs mutualisés fréquemment utilisés qui hébergent plusieurs types de services qui peuvent n'avoir aucun lien entre eux. La raréfaction des adresses IPv4 accentuent encore plus ce phénomène de mutualisation des adresses. Nous pourrions nous contenter de lister les destinations fréquentes comme les serveurs de Google ou Facebook, mais garder les listes d'adresses à jour serait rapidement laborieux.

Les serveurs mutualisés nous empêchent également d'utiliser les enregistrements PTR [RFC 1033] associés aux IP, qui sont rarement explicites. L'utilisation des services de WHOIS [RFC 3912] fournis par les LIR (*regional internet registries*) [9] pose également problème à cause du nombre relativement réduit de numéros de système autonome (AS) qui brassent un gros pourcentage des IP adressables rendant l'activité des machines associées difficilement prédictibles.

9. Organisations à but non-lucratif qui, *inter alia*, administrent et enregistrent les blocs d'adresses IP par région du monde.

Si la destination est une information intéressante, l'utilisation des fréquences peut être un complément intéressant.

2.2.4 Fréquences

L'observation des fréquences permet de résoudre deux problèmes :

1. dans le but d'inférer une localisation, nous pourrions supposer qu'un utilisateur a ses habitudes et qu'il utilise fréquemment les mêmes services depuis le même type d'endroit. Par exemple un utilisateur lambda pourrait visiter le site de la BBC chaque fois qu'il prend les transports en commun, Wikipédia lorsqu'il est dans un café pour appuyer ses arguments ou consulter son courriel professionnel quand il est dans ses heures de travail. Nous pourrions donc essayer de détecter des séquences d'adresses de destination et vérifier par l'expérimentation si elles peuvent être associées à des lieux particuliers ;
2. les écarts dans les fréquences (qu'on pourrait aussi qualifier d'irrégularité dans le rythme) pourraient aussi être des informations importantes. Dans la plupart des cas, les utilisateurs d'ordiphone ne consultent pas leurs courriels de façon active, mais les laissent se faire récupérer de façon automatique en demandant à leur logiciel de tourner en tâche de fond pour vérifier régulièrement la queue de messages du serveur distant. Ils les consultent ensuite en mode hors-ligne, lorsqu'ils sont notifiés. Par conséquent, nous ne pouvons pas associer ces flux (vers ou depuis le port IMAP 993 par exemple) à un comportement de l'utilisateur. Mais quand un courriel est consulté en mode hors-ligne, le protocole IMAP met à jour un drapeau sur le serveur pour indiquer le courriel a été lu, en utilisant le même port que pour la récupération automatique. Si durant la phase d'apprentissage nous arrivons à observer une fréquence de récupération des courriels (donc de connexion à un même serveur avec un port IMAP), nous pouvons aussi observer les irrégularités dans le rythme qui trahissent une action de l'utilisateur. Cette façon de faire permettrait de résoudre le problème de la routine en tâche de fond dans le cas du courriel, en réussissant malgré tout à détecter le moment où c'est réellement l'utilisateur qui est la source du flux. Si durant la phase d'apprentissage on ne constate aucune régularité dans la consultation des courriels, nous pourrions directement conclure que les flux à destination de ce couple serveur/port peuvent refléter une activité. Nous verrons dans la partie résultats que cette théorie est mise à mal par le système Android.

2.2.5 Dates et tailles

Les tailles des flux pourraient caractériser l'activité, et les dates pourraient être utilisées au minimum pour détecter les interruptions significatives dans l'usage du périphérique, et donc potentiellement un changement de lieu.

2.2.6 Synthèse

Le principal objectif de notre travail de recherche est de déterminer s'ils y a des séquences fréquentes et significatives d'adresses de destination qui pourraient caractériser une activité et donc un type de lieu. Les séquences d'adresses seront donc nos patrons et pourront potentiellement être discontinues. Puisque ces séquences seront probablement différentes d'un utilisateur à l'autre, elles doivent être déterminées via une phase d'apprentissage spécifique aux utilisateurs.

Une fois que les séquences sont trouvées, la second étape sera de les retrouver dans un autre jeu de données provenant du même utilisateur. Finalement, nous seront capables de déterminer si les périodes détectées comme correspondant plus ou moins à l'une de nos séquences correspondent ou non à des types de lieux spécifiques ou non.

Afin d'être proche d'un contexte de FAI, nous nous empêcherons de demander des informations aux utilisateurs qu'un opérateur de réseau ne pourrait obtenir sans son accord. Dans la mesure où nous n'avons pas accès à un vrai FAI mobile, les NetFlows seront récupérés directement depuis les périphériques grâce à Flowoid v2. Les données géotagguées et le nom des applications associées aux flux ajoutées par ce dernier ne seront utilisées que pour faire la vérification qui consistera à

catégoriser manuellement les lieux visités pour les comparer aux périodes détectées comme étant un type de lieu particulier.

2.3 Modélisation et analyse

En partant d'un jeu de NetFlows récupérés depuis l'activité réelle d'une utilisateur d'ordiphone équipé de Flowoid v2 et sur une période suffisamment longue (plusieurs semaines), les premières étapes sont :

1. élaguer les NetFlows proposés en entrée ;
2. trouver les séquences récurrentes d'adresses IP de destination (*sequencing*) en prenant soin de distinguer les nouvelles séquences des variations des précédentes ;
3. élaguer les séquences trouvées et leurs variations régulièrement ;
4. regrouper les séquences trouvées (*clustering*) pour obtenir la séquence la plus représentative de chacun des groupes ;
5. rechercher les séquences dans le même jeu de données et valider manuellement si les périodes qui semblent correspondre à une séquence particulière représentent bien un type de lieu ou non.

Nous considérons que les données en entrée pris depuis le collecteur sont sous la forme :

```
DATETIME DURATION SRC_IP DST_IP SIZE
DATETIME DURATION SRC_IP DST_IP SIZE
[...]
```

Dans un premier temps, seules les adresses IP de destination sont prises en considération.

2.3.1 Élagage des flux bruts

Un premier élagage des données brutes est effectué dans le but de réduire un maximum les NetFlows qui parasitent le jeu de données. Nous séparons également le jeu en plusieurs sous-jeux dès qu'une période trop longue est détectée entre la fin d'un NetFlow (début + durée) et le début du NetFlow qui suit, évitant ainsi de trouver des séquences entre leurs adresses alors que ça n'est manifestement pas la même activité. Enfin, nous supprimons les NetFlows relatifs à des requêtes DNS qui sont considérées comme du bruit, ainsi que les NetFlows avec des champs significatifs identiques qui se suivent et qui ont été séparés probablement à cause des temps d'expiration (les durées sont additionnées pour ne recréer qu'un seul flux).

2.3.2 Découverte des séquences

Nous appliquons ensuite notre propre DVSM (*discontinuous varied-order sequential miner*), l'algorithme de découverte des séquences fréquentes. Cette partie du modèle est basée sur un travail de Rashidi et al. [11] qui appliquent leur algorithme à un réseau de capteur dans un appartement intelligent. Nous avons largement adapté leur solution pour aboutir sur un algorithme plus réaliste (notamment en terme de traitement des données) et mieux adapté à notre cas.

Le modèle que nous utilisons parcours l'ensemble des séquences d'adresses qui peuvent être formées depuis le jeu de données. Pour ce faire, nous utilisons une fenêtre glissante en trouvant les suites de deux adresses, puis trois, quatre, et ainsi de suite comme illustré dans la figure 2.1. Chaque fois qu'une nouvelle séquence est trouvée, nous la comparons avec l'ensemble des séquences précédemment trouvées. Pour déterminer si une nouvelle séquence est la variation d'une précédente, nous utilisons la distance d'édition de Levenshtein [12]. La distance d'édition Δ représente le nombre d'éditions (e.g. insertion, suppression ou remplacement d'une adresse) requises pour transformer une séquence en une autre.

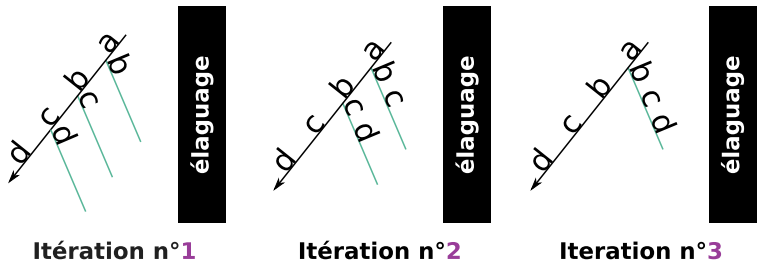

FIGURE 2.1 – Fenêtre glissante croissante.

Nous évaluons la similarité Θ entre deux séquences de cette façon :

$$\Theta(\alpha, \beta) = 1 - \frac{\Delta(\alpha, \beta)}{\max(\alpha, \beta)} \qquad \alpha \neq \beta \tag{2.1}$$

À la fin de chaque itération (représentant la taille de la fenêtre qui a parcouru le jeu de données en entier comme illustré dans la figure 2.1), nous élaguons les séquences et les variations qui sont insignifiantes afin de considérablement réduire le nombre de séquences découvertes et donc le temps d'exécution global (qui serait sinon purement exponentiel). Les deux métriques utilisées pour cela sont la taille de compression maximale dérivée du travail de Rissanen [13] et un comptage trivial de ses occurrences dans le jeu de données. Les variations sont stockées dans une table de hashage indexée par la séquence représentative.

Chacune des séquences ou variations qui ne valide pas ces deux contraintes est oubliée (`len` retournant le nombre d'adresses) :

$$\frac{\text{count}(seq) * \text{len}(seq)}{\text{len}(dataset)} < \lambda_1 \tag{2.2}$$

$$\text{count}(seq) < \lambda_2 \tag{2.3}$$

Dans un même temps, cette phase d'élagage se charge d'élire la séquence la plus représentative pour chaque groupe de variations. La sélection cherche la séquence qui est la plus proche par rapport à toutes les autres (moyenne des distances de Levenshtein). Si la séquence élue fait partie des variations actuelles, la séquence représentative devient une variation et la séquence élue devient la représentative. De cette façon, les index de la table de hashage représentent toujours les séquences la plus importantes ce qui est primordial pour l'étape de regroupement qui suivra.

L'élection de la séquence G la plus représentative se fait de la façon suivante (pour chaque itération, p est le candidat choisi parmi toutes les variations et la séquence actuellement représentative, v_i sont les autres choix et n_v représente le nombre de variations plus un) :

$$G = \arg\max_p \frac{\sum_{i=0}^{n_v-1} \Theta(p, v_i)}{n_v} \tag{2.4}$$

Nous continuons l'élagage en étendant les séquences jusqu'à ce qu'il n'y ait plus de séquence trouvée qui ne soit pas la variation d'une précédente.

2.3.3 Élagage des modèles

L'étape précédente nous a permise d'obtenir une liste de séquences fréquentes avec leurs dérivées. Mais certaines de ces séquences ne sont pas importantes, ce pourquoi un nouvel élagage général doit être effectué.

Deux stratégies sont utilisées pour nettoyer notre liste :

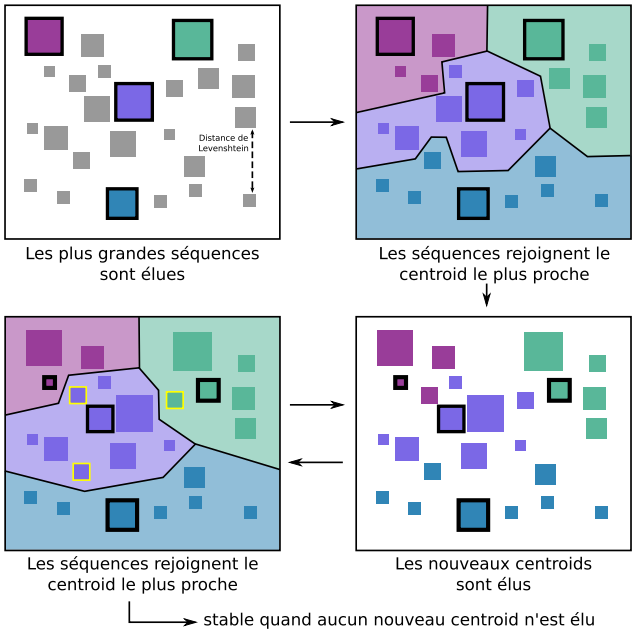

FIGURE 2.2 – Regroupement par les K-means.

1. la première consiste à supprimer toutes les séquences qui n'ont pas de variation, en considérant qu'il n'est statistiquement pas possible qu'une séquence importante et fréquente ne soit jamais parasitée par d'autres flux ;
2. la seconde consiste à maximiser les séquences, en supprimant toutes les séquences qui sont contenues dans d'autres séquences (en les préfixant ou en les suffixant).

2.3.4 Regroupements

Même si la liste des séquences trouvées est maintenant réduite, nous ne pouvons pas garder en garder des dizaines puisqu'il faudrait alors caractériser autant de types de lieu. C'est pourquoi notre objectif sera plutôt d'arriver à une réduction qui se limite à quatre séquences, correspondant à quelque chose comme *maison*, *travail*, *café* et *rue*. Pour arriver à ce niveau de réduction, nous utilisons une méthode de regroupement basée sur l'algorithme des K-means [14] comme illustré dans la figure 2.2.

Nous commençons par initialiser quatre centroïds (ou *means*) avec les quatre séquences les plus grandes (cette dernière étape ne considère que les séquences représentatives qui servent d'index à la table de hashage, et ignore totalement les variations). Nous préférons cette méthode plutôt que celle à base d'aléatoire, qui est en général favorisée, pour privilégier l'émergence des séquences les plus grandes possibles afin de couvrir les périodes les plus longues possibles lors de leur utilisation. Puis nous associons chacune des autres séquences au centroïd le plus proche d'après notre fonction de similarité $\Theta(\alpha, \beta)$, créant ainsi quatre groupes. Enfin, pour chacun des groupes, nous élisons la séquence qui est la plus proche de toutes les autres séquences du groupe en tant que nouveau centroïd.

Ces trois étapes sont répétées jusqu'à ce que l'élection de tous les centroïds renvoie deux fois le même résultat, indiquant la stabilité du groupe. À la fin de l'élection, nous obtenons les quatre séquences, supposées correspondre à des habitudes et donc potentiellement à des types de lieu.

2.3.5 Détection des séquences

Une fois que la phase d'apprentissage est achevée, nous obtenons nos séquences (e.g. quatre séquences d'adresses IP) spécifiques à l'utilisateur qui est à l'origine du jeu de données initial. Le but est maintenant de reconnaître ces séquences dans le même jeu de données et de déterminer les périodes de son activité qui leur correspondraient.

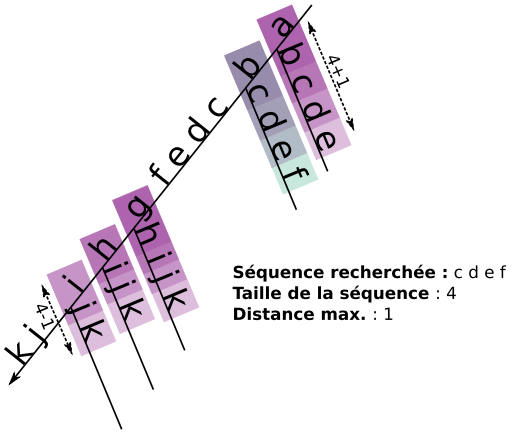

FIGURE 2.3 – Détection avec le *backtracking*.

Pour cette dernière étape dans la validation de notre processus, nous utilisons une sorte de *backtracking* [15], comme illustré dans la figure 2.3. Nous utilisons de nouveau une fenêtre glissante pour parcourir les adresses du jeu de données mais cette fois-ci avec une unique itération. Pour chaque adresse, nous ajoutons les `maxPatternSize + maxDistanceAuthorized` adresses suivantes. Et pour chaque ajout d'adresse nous comparons la séquence en cours avec les quatre séquences trouvées lors de la première phase, de la plus petite à la plus grande. Si plusieurs séquences sont détectées alors la plus grande est choisie pour représenter la période. Si au moins une séquence correspond, nous enregistrons la date et l'heure de début du NetFlow correspondant à la première adresse de la séquence en cours ainsi que la date et l'heure de fin associées au NetFlow correspondant à la dernière adresse auxquelles nous ajoutons la durée de ce dernier. L'identifiant de la séquence trouvée est également enregistré. De cette façon, nous obtenons une période supposée correspondre à un type de lieu spécifique représenté par l'identifiant de l'une de nos quatre séquences. La boucle continue à partir de l'adresse qui suit la dernière adresse de la séquence complète qui vient d'être identifiée.

Si aucune des quatre séquences n'a été identifiée dans la séquence courante la plus grande, la boucle continue avec l'adresse qui suit en oubliant tous ceux qui ont composé la tentative précédente.

La figure 2.3 montre un exemple avec la liste d'adresses `a b c d e f g h i j k` et seulement une séquence à trouver. Puisque la distance d'édition maximale est 1 dans cet exemple, nous cherchons jusqu'à quatre adresses après l'adresse courante. Quand nous parcourons la seconde adresse, après avoir vainement essayé avec `b c`, `b c d` et `b c d e`, nous pouvons voir que la séquence `b c d e f` correspond à notre séquence à trouver `c d e f` (en utilisant la distance d'édition maximale autorisée). Le processus continue avec l'adresse `f` qui suit. Il finit quand les séquences deviennent plus petites que la séquence à trouver moins le nombre maximum de suppressions possibles correspondant à la distance d'édition maximale autorisée.

En se basant sur le modèle qui vient d'être présenté et cette méthode utile pour exploiter les résultats, nous pouvons maintenant comparer les périodes trouvées automatiquement avec de vrais changements de lieux. La comparaison se fera à partir de l'interprétation humain des vrais mouvements et permettra de conclure si oui ou non les résultats ont un sens.

Chapitre 3

Résultats

Sommaire

Cette section présente les expérimentations qui ont été réalisées pour valider le travail de recherche, ainsi que les résultats obtenus.

3.1 Expérimentations

3.1.1 Implémentation du modèle

Le modèle présenté dans les sections précédentes a été entièrement implémenté grâce à un script Perl, capable de prendre en entrée un jeu de données. Ce dernier est un export des NetFlows collectés par Android depuis le collecteur, sur une période choisie. Les différents jeux de données utilisés pour les expérimentations proviennent de l'activité de plusieurs utilisateurs d'ordiphone qui ont accepté de laisser Flowoid capter et synthétiser toute leur activité réseau sur des périodes allant jusqu'à trois semaines.

À partir d'un simple fichier texte représentant les NetFlows, l'implémentation du modèle permet de faire ressortir plusieurs séquences d'adresses de destination, censées pouvoir représenter un type de lieu. Le nombre de séquences par défaut est quatre, mais le script est entièrement paramétrable depuis la ligne de commande pour pouvoir observer automatiquement l'influence de chacune des variables du modèle sur les séquences trouvées. Au-delà de l'implémentation du modèle, le script permet de reparcourir le jeu de données en utilisant les séquences préalablement trouvées pour faire ressortir les périodes du jeu de données qui pourraient leur correspondre (à une distance d'édition près).

3.1.2 Validation des résultats

L'évaluation de la pertinence des séquences trouvées ne pouvant pas être automatisée, elle a été évaluée de façon empirique à partir des périodes qui ressortent et de l'identifiant de la séquence associée. L'expérimentation est jugée réussite si, pour les périodes du jeu de données qui sont labellisées avec le même identifiant, le type de lieu est effectivement identique. Il faut également que ce type de lieu ne se retrouve pas à d'autres instants sans qu'il n'ait été repéré. On admet un niveau d'imprécision le cas échéant, mais il faut donc pouvoir le quantifier pour obtenir une évaluation pertinente du modèle.

Sur des jeux de données qui varient de 30 000 à 60 000 flux, la première étape est donc de réussir à classifier manuellement les lieux en fonction des périodes. Si les données additionnelles ajoutées par Flowoid (coordonnées géographiques et nom d'application associé) ont été une aide précieuse, cette étape indispensable pour pouvoir comparer la réalité avec les résultats aura été particulièrement fastidieuse et délicate. À l'aide de différents scripts, le jeu de données a été réduit à une liste de coordonnées géographiques. Chacune a ensuite dû être soumise au service Google Maps pour visualiser le lieu en question. De ce lieu géographique, il a fallu deviner l'activité du contributeur pour pouvoir la classifier, parfois en lui demandant directement « que faisais-tu ici à cette heure-ci ? ». Il a donc s'agit de longues heures à retracer la vie privée des utilisateurs en les interrogeant régulièrement.

Outre la nécessité de ne pas être complexé devoir interroger des gens sur leur vie privée, ces derniers avaient parfois du mal à se souvenir de leur activité à ce moment-là. La seconde grande difficulté est que Flowoid propose plusieurs méthodes pour géolocaliser le téléphone, qui vont d'une méthode totalement passive à l'activation du GPS. Afin de préserver la batterie du mobile, la plupart des contributeurs ont choisi la première méthode. Celle-ci ne demande jamais explicitement la localisation du téléphone, mais consulte la dernière localisation qui a été demandée par une autre application, quelque soit la méthode utilisée (réseau ou GPS). À cause des localisations qui ne sont parfois pas à jour et de l'imprécision de méthodes comme l'utilisation du réseau, les coordonnées géographiques associées aux flux étaient parfois très éloignées du lieu réel. Par exemple, il est donc impossible de savoir précisément si l'utilisateur est dans sa voiture dans le quartier de son lieu de travail, ou s'il est déjà arrivé et installé à son bureau (ce qui devrait pourtant être vu comme deux types de lieu distincts, avec une activité différente).

Une fois toutes les coordonnées géographiques identifiées, une nouvelle commande automatisée a permis d'obtenir une liste de période regroupées par type de lieu, pouvant servir de témoin. Celle liste a été comparée aux périodes trouvées automatiquement.

3.2 Pertinence des résultats

3.2.1 Des résultats décevants

Après bien des essais en modifiant les variables du modèle et en essayant de lancer des batteries de tests dans le but d'obtenir des graphiques censés permettre de visualiser les meilleures valeurs, aucun résultat significatif n'a été trouvé. Même en s'appuyant sur l'aspect très imprécis du témoin, les changements de lieu que supposaient les périodes trouvées automatiquement étaient anarchiques.

Si le problème était bien visible en comparant avec les périodes témoins, il se laissait aussi supposer d'un seul coup d'œil. La nuit étant un indicateur précieux, il y a des heures où le propriétaire nous a assuré qu'il ne pouvait que dormir. Dans ces créneaux horaires, les périodes trouvées automatiquement indiquaient pourtant plusieurs changements d'activité et donc de lieu, qu'on retrouvait plus tard à différents moments de la journée. Les essais avec le jeu de données d'autres utilisateurs n'ont pas donné de meilleurs résultats.

Puisque le modèle supposait qu'un utilisateur ait une utilisation différente de son téléphone en fonction du lieu où il se trouve, les recherches auraient pu s'arrêter brutalement en supposant que cette supposition était fausse. Sans pour autant réussir à le prouver (ce qui aurait été en soit un résultat intéressant), et sans pouvoir affirmer que ça ne soit pas l'implémentation du modèle plutôt que le modèle qui soit en cause.

3.2.2 Bruit de fond

En cherchant à mieux comprendre le comportement du modèle utilisé sur les jeux de données, un détail a rapidement été remarqué au gré des différents tests. On le retrouve sur cette liste de quatre séquences trouvée sur un jeu de données d'environ 30 000 NetFlows :

```
[ #0 ] 69.171.230.22 193.51.193.146 178.170.95.127 193.51.193.146
       178.170.95.127 69.171.248.112 31.13.80.1 178.170.95.127
       193.51.193.146 178.170.95.127 69.171.248.112
[ #1 ] 178.170.95.127 98.137.200.255 188.125.73.190 178.170.95.127
       173.252.102.48 193.51.193.146 178.170.95.127 31.13.80.33
       178.170.95.127 173.194.78.188 178.170.95.127
[ #2 ] 178.170.95.127 193.51.193.146 178.170.95.127 69.171.248.112
       178.170.95.127 193.51.193.146 178.170.95.127 173.194.78.188
       178.170.95.127 31.13.80.33 31.13.81.65
[ #3 ] 178.170.95.127 193.51.193.146 178.170.95.127 31.13.80.33
       98.137.200.255 188.125.73.190 178.170.95.127 69.171.245.80
       173.194.78.188 193.51.193.146 178.170.95.127
```

On peut aisément constater que plusieurs adresses reviennent régulièrement. C'est le cas par exemple pour `178.170.95.127` qu'on retrouve parfois jusqu'à une adresse sur deux. Il s'agit de l'adresse du serveur de courriels qu'utilise le propriétaire du téléphone. En regardant le jeu de données avec les champs additionnels de Flowoid, dont le nom de l'application associée, on comprend rapidement qu'il s'agit d'une routine qui vérifie régulièrement les courriers sur le port IMAP 993. D'autres correspondent à Facebook et sont également le fruit d'une routine qui consulte régulièrement ces adresses.

La présence de ces routines dans les séquences est une pollution dans le cadre de nos recherches puisqu'elles ne dépendent pas de l'activité de l'utilisateur et encore moins du type de lieu dans lequel il se trouve. Par contre, constater qu'elles se retrouvent dans les séquences, qui sont censées être représentatives des activités régulières du téléphone, tend à prouver que le modèle fonctionne bien à ce niveau.

Le bruit de fond généré par les routines avait été discuté lors de la conception du modèle. En supposant que celui-ci est en permanence régulier, nous avions supposé qu'il serait neutre pour les séquences trouvées. Dans les sections précédentes, nous avions proposé d'utiliser les fréquences pour les détecter et ne garder que les écarts de fréquence qui devraient correspondre à une action de l'utilisateur. Puisque le modèle uniquement basé sur les adresses de destination ne semble pas satisfaisant, les recherches ont continué dans ce sens.

La figure 3.1 représente graphiquement le temps écoulé entre deux émissions de flux à destination du serveur de courriels sur le port IMAP. On observe parfaitement un centrage autour de quinze minutes, qui correspond effectivement à la fréquence réglée par l'utilisateur pour faire vérifier son courrier. On observe également quelques écarts de fréquence, qui pourraient être le fruit d'une activité d'utilisateur (lecture d'un courrier, envoi d'un courrier, etc.).

Manuellement, chacun de ces pics a été associé à une date et une heure. Une liste a ainsi été composée et a été soumise à l'utilisateur pour lui demander d'en juger la pertinence. Malheureusement, même si certaines heures pourraient correspondre, ce dernier en a très rapidement pointé d'autres qu'il jugeait totalement improbables (encore une fois grâce aux heures de nuit durant lesquelles il est certain de ne rien avoir fait sur son téléphone).

La figure 3.1 a été manipulée après la première observation. Beaucoup de pics étaient à trente minutes, et certains étaient des écarts de moins de une minute. Étant donné le nombre important de pics en multiple de quinze, nous avons jugé que le logiciel « ratait » parfois des vérifications. Les multiples ainsi que les écarts de moins de trois minutes ont été ramenés à quinze. Malgré cet effort, les pics ne sont toujours pas représentatifs, l'imprécision du téléphone étant beaucoup trop grande comparé à des intervalles de seulement quinze minutes. Cette imperfection est formalisée par la plateforme Android, puisque ce dernier permet de programmer des vérifications à des temps inexacts. L'objectif est de laisser le téléphone gérer tout seul les déclenchements de routine, pour les rassembler et faire en fonction du réseau et de l'état de la batterie.

Du côté de Facebook, la figure 3.2 montre que les écarts entre les flux sont interprétables.

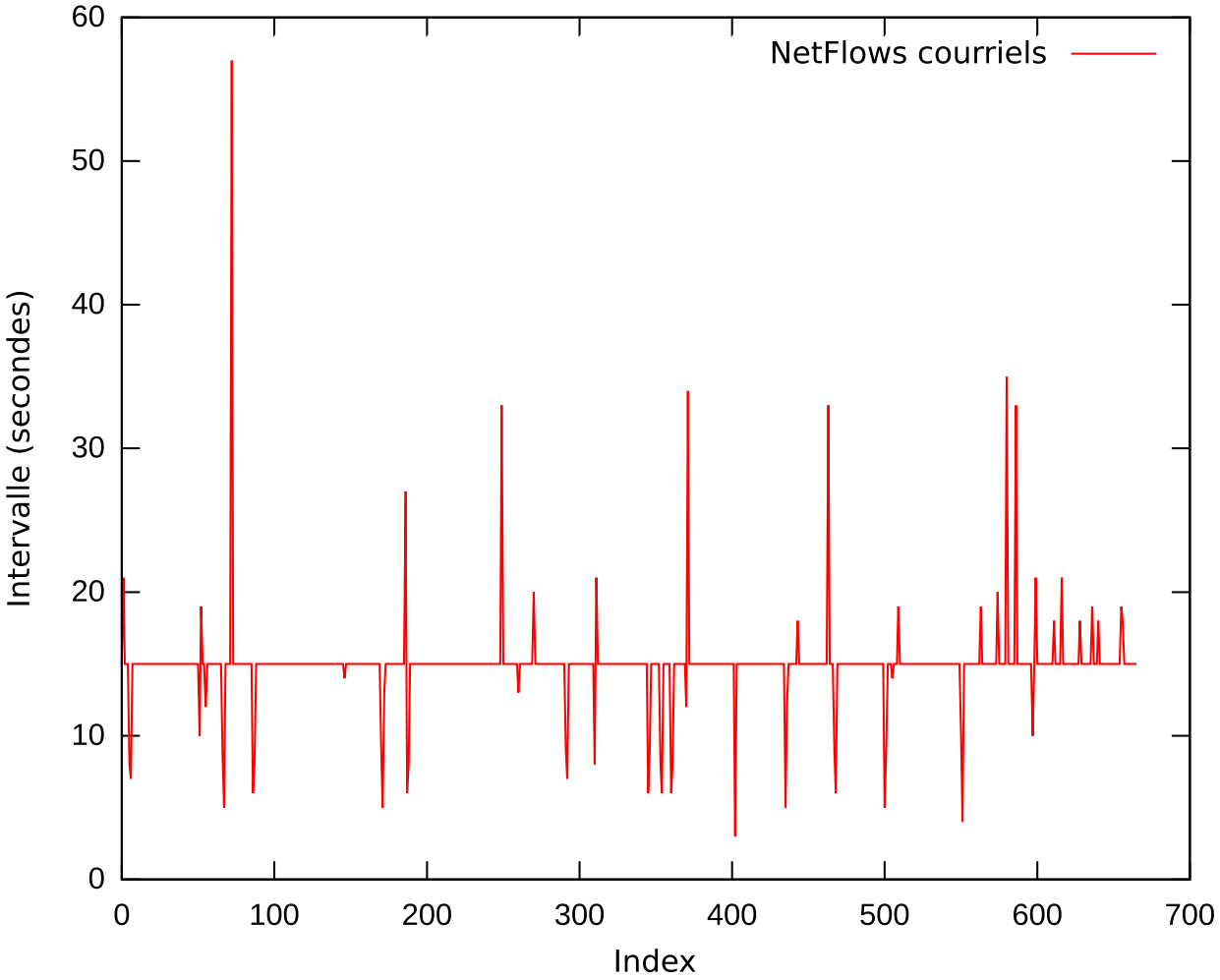

FIGURE 3.1 – Intervalle de temps entre deux NetFlows de vérification des courriels (port 993).

Les autres métriques accessibles depuis les NetFlows (nombre de paquets et durée du flux) sont encore moins significatives pour distinguer les activités automatiques des activités de l'utilisateur. Un exemple est donné en figure 3.3.

Détecter le trafic des routines sur un jeu de données à partir des fréquences semble impossible. Et utiliser notre modèle pour détecter les activités régulières de l'utilisateur avec ce bruit de fond semble compromis.

3.3 Un résultat peut en cacher un autre

Malgré la déception, l'investigation sur la compréhension de ces séquences a été poursuivie. En reprenant toutes les adresses une à une, il a semblé soudainement probable que l'intégralité des séquences soient composées de routines. Si nous avons vu dans la section précédente que les routines n'ont pas une régularité très précise, elles ne pourront jamais être moins régulière que la plus régulière des activités de l'utilisateur. En réalité, le modèle que nous avons conçu est condamné à détecter le trafic des routines du téléphone.

Pour les séquences données en exemple ci-dessus, on trouve :

```
18 mailperso.org.
11 facebook.com.
 8 inria.fr.
 4 yahoo.com.
 3 1e100.net.
```

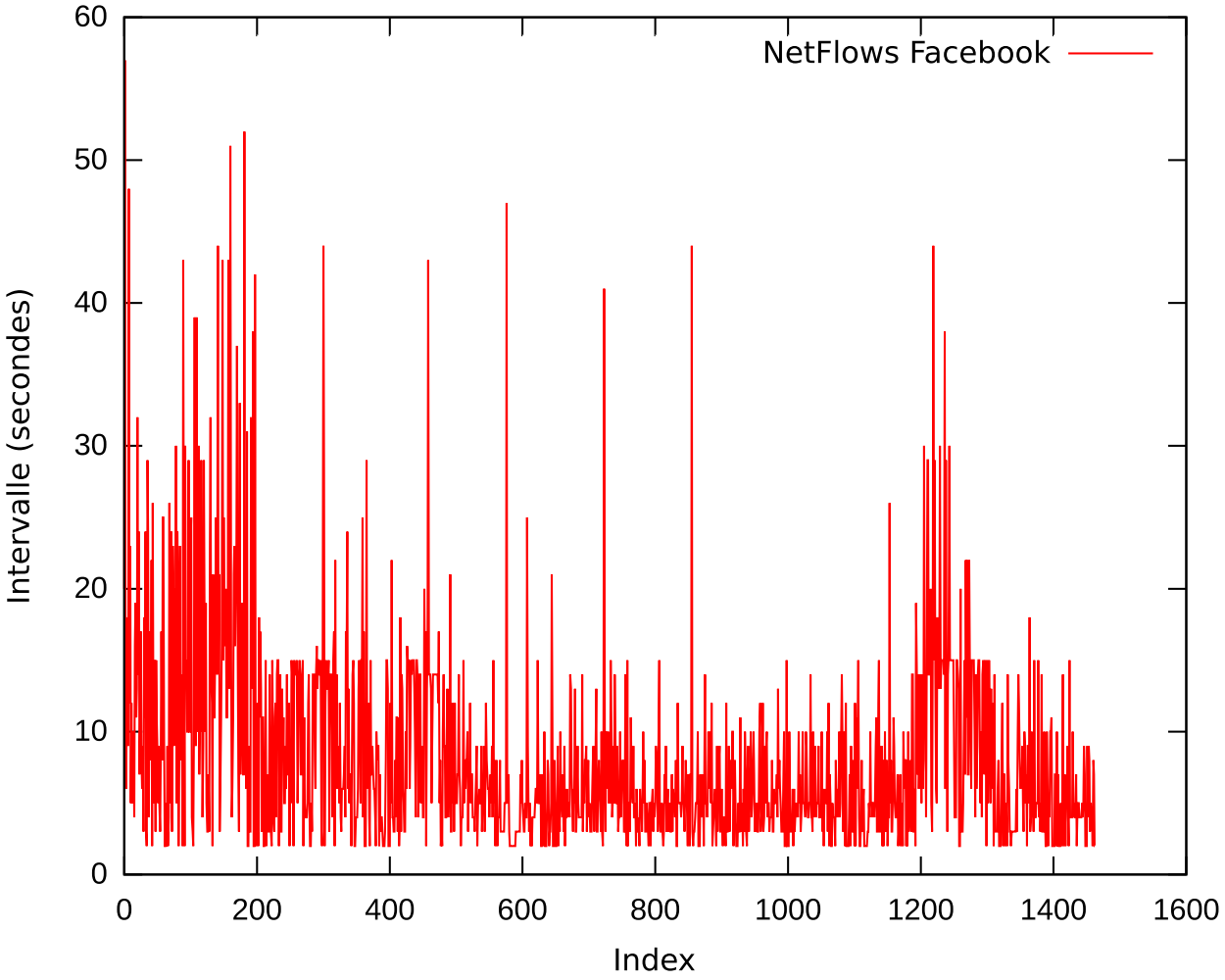

FIGURE 3.2 – Intervalle de temps entre deux NetFlows à l'initiative de l'application Facebook.

Soit 59% uniquement consacrés aux mails (vérification régulière), 25% à Facebook (synchronisation), 9% à Yahoo (météo) et presque 7% à Google (cadre de travail Android).

Pour aller plus loin, nous avons décidé de vérifier jusqu'à quel point cette nouvelle supposition est vraie. Nous avons donc repris le jeu de données initial et nous avons supprimé tous les flux qui communiquent avec ces adresses. En observant rapidement ce qui reste et grâce au champ application de Flowoid, nous avons constaté qu'il ne restait plus que du trafic de surf, et d'applications qui ne faisaient pas de routines. Il restait également du Facebook, mais avec un nombre de flux suffisamment grand pour supposer que ça ne puisse pas être de la routine.

La figure 3.4 représente le nombre de NetFlows à chaque instant sur un jeu de données de dix jours. Les flux verts représentent le trafic qui communique avec des adresses de nos séquences et les rouges le reste du trafic imputable à l'utilisateur. Le nombre de flux peu élevé et surtout sa régularité dans le temps ressemble à ce qu'on pourrait attendre d'un bruit de fond généré par des routines, alors que les deux types de NetFlow ont été séparés du début à la fin de façon entièrement automatique. Le trafic rouge est beaucoup plus anarchique et montre un nombre élevé de flux simultanés, ce qui semble cohérent pour du trafic d'utilisateur imprévisible et gourmand. À noter qu'à cause des temps d'expiration de Flowoid le nombre de NetFlows doit être considéré de façon relative, la plupart des flux ayant probablement terminés plus tôt que ce que le graphique représente.

Si le modèle semble particulièrement bien fonctionner pour séparer les deux types de trafic, il n'est certainement pas parfait. Ne serait-ce que parce qu'on constate trop rarement une baisse significative du trafic d'utilisateur qui pourrait correspondre aux nuits. Les services comme Google et Facebook posent également problème parce qu'ils changent régulièrement d'adresse. Ainsi, pour un utilisateur qui a beaucoup d'applications, les expériences ont prouvé qu'il fallait augmenter le

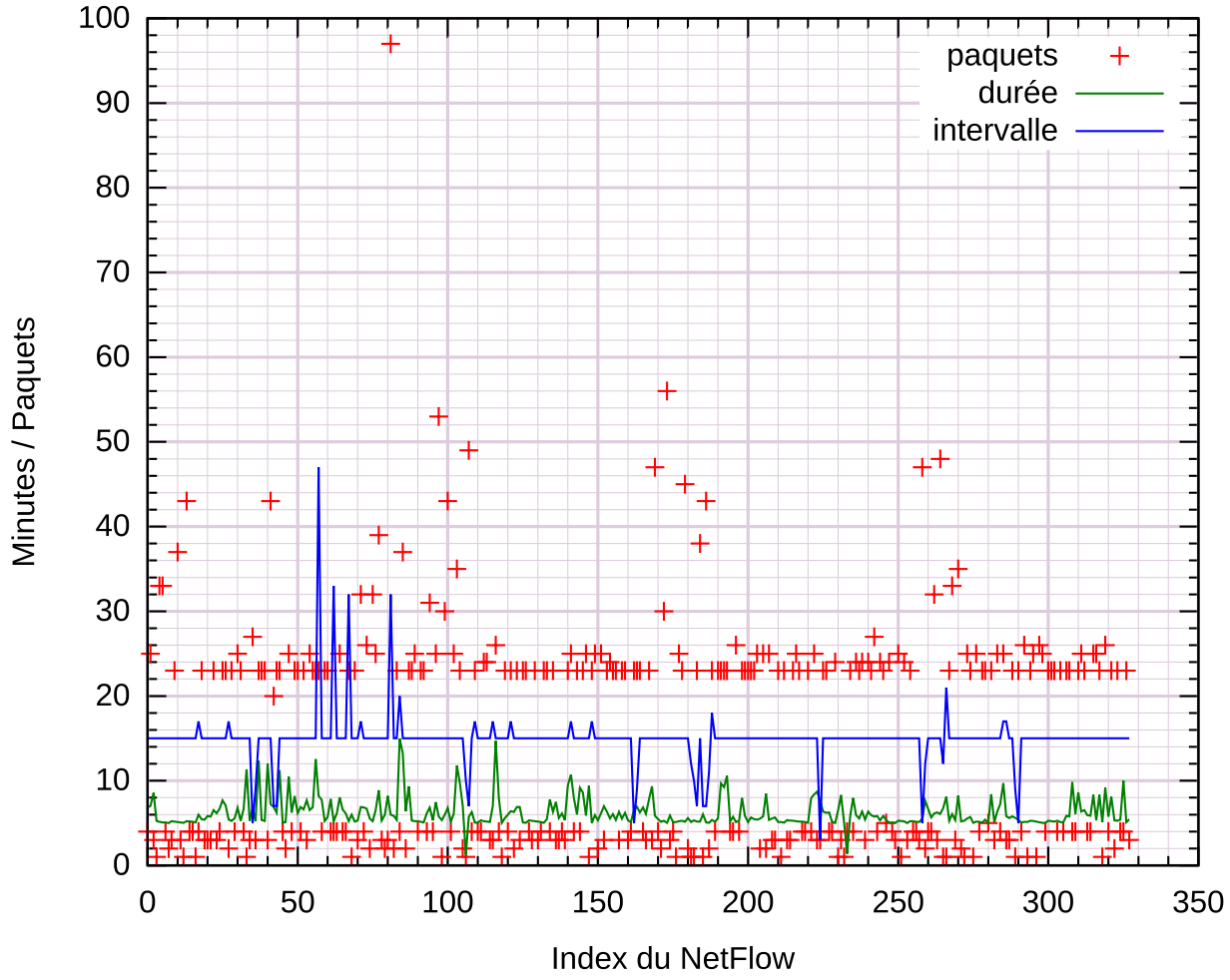

FIGURE 3.3 – Comparaison des différentes métriques pour des NetFlows de vérification des courriels (port 993).

nombre de séquences à trouver pour que ces dernières intègrent un maximum d'adresses différentes de routine, pour obtenir un résultat similaire.

Pour valider ce résultat de façon plus stricte, il faudrait avoir un jeu de données provenant du même téléphone avec l'assurance qu'il n'y a aucune activité d'utilisateur dessus. Le nouveau jeu de données pourrait alors être comparé à celui qui aura été filtré (dans le sens inverse) automatiquement en observant le pourcentage représenté par l'intervalle qui réuni les deux. Il pourrait également être représenté graphiquement pour voir si son trafic ressemble à celui de la courbe verte de la figure 3.4.

Cette expérience devrait être effectuée à la suite du stage, n'ayant plus dix jours au moment des résultats pour acquérir le nouveau jeu de données. Par ailleurs, il aurait plutôt fallu le double de temps dans la mesure où l'utilisateur qui a fourni le premier jeu de données n'est pas prêt à ne pas toucher à son téléphone durant dix jours. Trouver un utilisateur intensif d'ordiphone qui est prêt à ne pas l'utiliser sur une grosse période de temps semble d'ailleurs suffisamment contradictoire pour que l'expérience ne soit pas si simple à mener à bien.

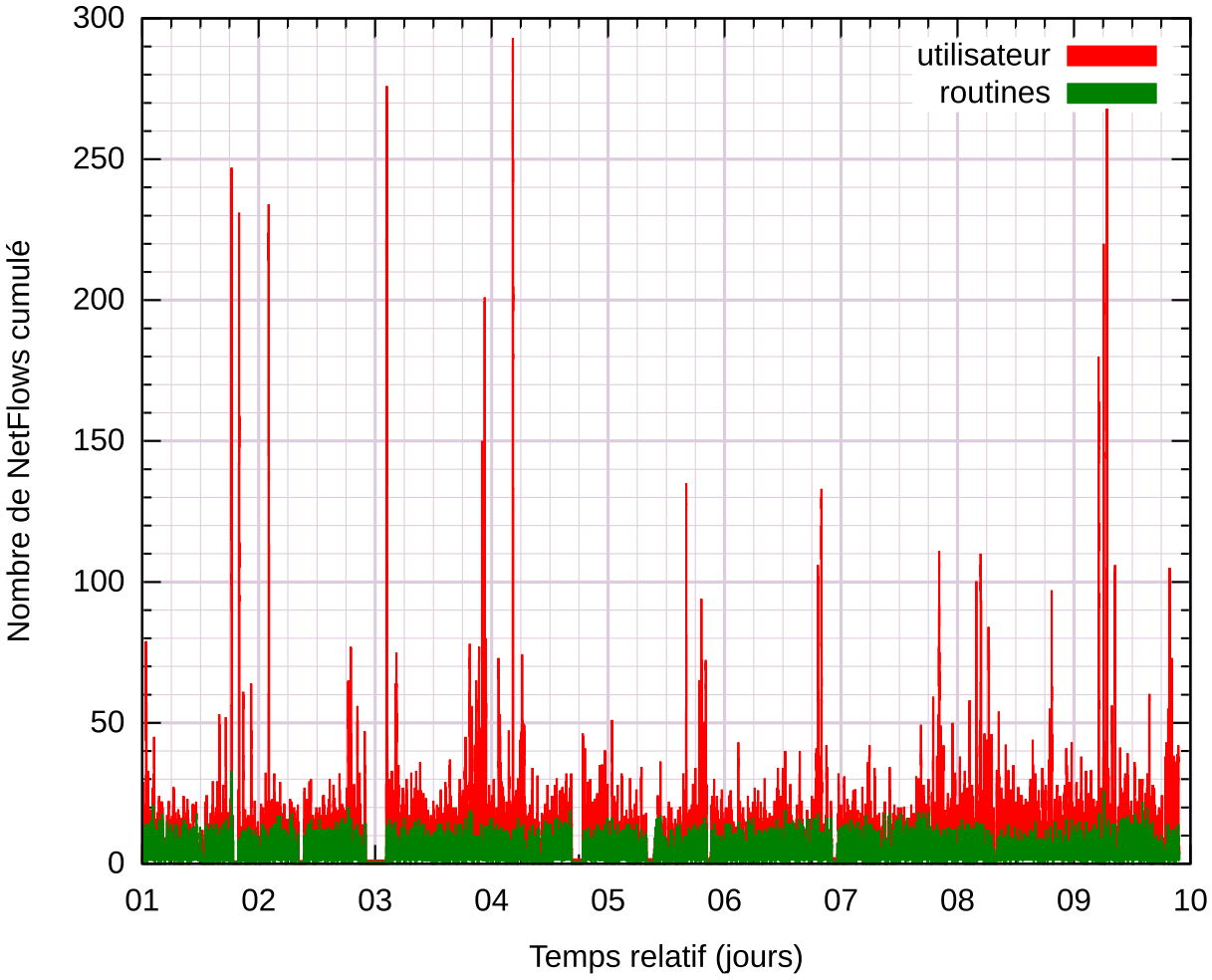

FIGURE 3.4 – Séparation entre le trafic supposé des routines et des actions de l'utilisateur.

Conclusion

Ingénierie

Même si quelques problèmes sont encore à déplorer avec Flowoid v2 sur quelques anciennes versions de Android (déconnexion du pilote probablement dû à une politique de nettoyage des processus assez sévère), le logiciel va au-delà des attentes initiales.

Grâce à sa bibliothèque qui se veut le parfait reflet Java de la RFC 3954, l'exportateur de NetFlows est parfaitement compatible avec la version neuf du format NetFlows. Ainsi, il exploite tout le potentiel du système de modèles et permet d'exporter tout type d'information simplement en ajoutant des classes courtes et quelques lignes de code, avec une interface de programmation simple, claire et efficace. Grâce à la séparation de l'exportateur en bibliothèque abstraite, n'importe qui peut créer un logiciel Android qui exporter des NetFlows, sans se contraindre à utiliser le travail plus spécifique qui a été fait pour Flowoid v2. Avec le système de pilotes, l'exportateur est parfaitement modulaire, de la capture des informations à leur export.

Le logiciel Flowoid v2 quant à lui, propose un parfait exemple d'implémentation de la bibliothèque. Certaines fonctionnalités simples comme l'export des coordonnées géographiques ont été implémentées, et quelques autres plus délicates comme l'association avec le nom du processus ont été ajoutées. L'interface graphique proposée permet aux utilisateurs finaux de visualiser en temps réel l'activité de l'exportateur et de le configurer précisément pour adapter son comportement.

Enfin, grâce à la réécriture de l'ensemble des composants de Flowoid v1, en plus d'offrir plus de fonctionnalités et de flexibilité, la bibliothèque et son implémentation pourront maintenant être fournies par Inria sous une licence libre GPL v2 [1].

Recherche

Si les résultats pour l'aspect recherche ne sont pas encore totalement concluants, le travail qui a été effectué aura constitué une excellente première approche du travail de chercheur.

Le travail sur la géolocalisation qui a été initialement proposé était un défi d'une ampleur inhabituelle. Il n'était pas absolument acquis que les activités d'un utilisateur d'ordiphone puissent être spécifiques à un type de lieu (mon encadrant et moi-même avions d'ailleurs le sentiment que ça n'était pas le cas dans la majorité des cas), ni que l'activité réseau à elle-seule puissent suffisamment les caractériser.

Malgré tout, la tentative qui a été faite a été très intéressante à produire. Mieux, elle a finalement abouti à l'espoir d'un autre résultat, avec la différenciation des flux de routines et d'utilisateur. En terme de recherche, savoir chercher en acceptant que la supposition de départ puisse être erronée , comme réussir à découvrir un résultat inattendu est particulièrement formateur. D'autant plus que ce résultat est finalement beaucoup plus intéressant que celui qui était visé, puisque tous les travaux de recherche qui souhaitent se baser sur l'activité réseau de l'utilisateur se retrouveront confrontés au même problème de bruit de fond. Si ce résultat est confirmé et qu'il fait l'objet d'un

1. La bibliothèque pourrait être fournie en GPL v3, mais pas Flowoid à cause du parseur Java de Netutils qui n'est pas disponible dans cette version de la licence. Une demande à ses auteurs a été faite pour savoir s'ils accepteraient de le distribuer en GPL v3.

FIGURE 3.5 – Représentation des serveurs contactés par un utilisateur d'ordiphone avec SurfMap (avec une forte agrégation).

papier de recherche, nous sommes donc en droit d'espérer qu'il soit cité par d'autres travaux, signe de reconnaissance dans le domaine. Un premier papier de recherche de six pages en anglais sur la géolocalisation a été produit durant la phase de collecte des jeux de données, pour la conférence WPES à Berlin. Si ce papier ne pourra finalement probablement pas être utilisé en l'état pour une publication, il m'aura permis d'apprendre à appréhender ce type de travail essentiel.

Perspectives

Flowoid est actuellement le seul exportateur de NetFlows pour Android qui existe. Étant délivré sous licence libre, il aura donc l'espoir d'être réutilisé dans beaucoup d'autres travaux, grâce à sa modularité extrême. Un premier exemple d'exploitation est déjà disponible, puisque notre correspondant de l'Université de Twente utilise déjà Flowoid pour tracer le déplacement d'un utilisateur d'ordiphone et représenter les serveurs qui sont contactés (voir les captures d'écran de l'outil SurfMap en figures 3.5 et 3.6). Une présentation dans un groupe de travail de l'IETF sera faite le 28 juillet 2013 à Berlin, pour présenter Flowoid et quelques mesures statistiques sur l'utilisation du réseau par un ordiphone, qui ont été réalisées par Abdelkader LAHMADI à partir des données collectées.

Les résultats de recherche sont finalement très encourageants. Puisque les résultats n'étaient pas ceux attendus, nous avons simplement manqué de temps pour aller jusqu'au bout des expérimentations qui nécessitent de grosses périodes de collecte de données. Cependant, mon travail sera repris suite à mon stage pour qu'il aboutisse finalement sur un papier de recherche, basé sur celui que j'ai écrit pour la géolocalisation.

Ces quelques mois au sein de l'équipe Madynes auront finalement correspondu à mes attentes. Après avoir validé mes compétences d'ingénieur, j'ai pu m'essayer à la recherche en touchant de près à ses principaux aspects. Les échanges avec mon encadrant, les autres stagiaires ou doctorants ont été très enrichissants. J'ai pu me faire une idée claire du monde de la recherche, ce qui était mon principal objectif en faisant mon stage dans le laboratoire. Le second objectif, dès lors que j'ai confirmé mon intérêt pour ce type de travail, a été de chercher une solution pour continuer mes études en doctorat.

FIGURE 3.6 – Représentation des serveurs contactés par un utilisateur d'ordiphone avec SurfMap (avec une agrégation moindre).

Après avoir discuté avec plusieurs permanents, j'ai finalement été contacté par Laurent CIARLETTA (Madynes) qui m'a proposé une thèse en partenariat avec l'équipe MAIA et EDF R&D. Après plusieurs entretiens avec Laurent et le futur directeur de thèse Vincent CHEVRIER (MAIA), un important échange de documents et un accord de la direction, je suis maintenant assuré de revenir au laboratoire pour trois ans, dès le quinze septembre.

Bibliographie

[1] R. Clarke, "Deep Packet Inspection : Its Nature and Implications", commissioned as a contribution to a 'Collection of Essays from Industry Experts' on Deep Packet Inspection by the Privacy Commissioner of Canada. En ligne : http://www.rogerclarke.com/II/DPI08.html.

[2] A.G. Mason, *Cisco Secure Virtual Private Network*, Cisco Press, 2002, p. 7.

[3] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, B. Stiller, "An overview of ip flow-based intrusion detection", *Communications Surveys Tutorials*, IEEE 12, March 2010, pp. 343–356. En ligne : http://wwwhome.cs.utwente.nl/~sperottoa/papers/2010/flow_based_id.pdf

[4] G. Vliek, "Detecting spam machines, a Netflow-data based approach", Thesis, February 2009. En ligne : http://essay.utwente.nl/58583/1/scriptie_G_Vliek.pdf

[5] H. Weststrate, "Botnet detection using netflow information - Finding new botnets based on client connections", *Structure*, 2009. En ligne : http://referaat.cs.utwente.nl/conference/10/paper/6935/botnet-detection-using-netflow-information.pdf

[6] B. Li, J. Springera, G. Bebisb, M.H. Gunesb, "A survey of network flow applications", *Journal of Network and Computer Applications*, Volume 36, Issue 2, March 2013, pp 567–581. En ligne : http://www.sciencedirect.com/science/article/pii/S1084804512002676#bbib87

[7] N. Melnikov and J. Shönwälder, "Cybermetrics : User Identification through Network Flow Analysis", *Mechanisms for Autonomous Management of Networks and Services*, Lecture Notes in Computer Science Volume 6155, 2010, pp 167-170. En ligne : http://link.springer.com/chapitre/10.1007%2F978-3-642-13986-4_24?LI=true#page-1

[8] P. Minařík, J. Vykopal, "Improving Host Profiling with Bidirectional Flows", *International Conference on Computational Science and Engineering*, 2009. En ligne : http://www.researchgate.net/publication/220775805_Improving_Host_Profiling_with_Bidirectional_Flows

[9] M. Soysal, E.G. Schmidt, "Machine learning algorithms for accurate flow-based network traffic classification : Evaluation and comparison", *Performance Evaluation*, 67, June 2010, pp. 451–467. En ligne : http://www.sciencedirect.com/science/article/pii/S0166531610000027

[10] O. Festor, A. Lahmadi, "Information Elements for device location in IPFIX", IETF Internet Draft, July 2012. En ligne : http://tools.ietf.org/html/draft-irtf-nmrg-location-ipfix-00

[11] P. Rashidi, N. Krishnan, D.J. Cook. "Discovering and Tracking Patterns of Interest in Security Sensor Streams", *Securing Cyber-Physical Critical Infrastructure*, chapitre 19 pp. 481-504, 2012. ISBN : 9780124158153

[12] V.I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals", *Soviet Physics Doklady*, Volume 10, Number 8, Febuary 1966, pp. 707-710. En ligne : http://profs.sci.univr.it/~liptak/ALBioinfo/files/levenshtein66.pdf

[13] J. Rissanen, "Modeling by shortest data description", *Automatica*, Volume 14, pp. 465-471. 1978.

[14] J.B. MacQueen, "Some Methods for classification and Analysis of Multivariate Observations", *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*, University of California Press, 1967, pp. 281–297. En ligne : http://projecteuclid.org/DPubS?service=UI&version=1.0&verb=Display&handle=euclid.bsmsp/1200512992

[15] D.E. Knuth, *The Art of Computer Programming*, Addison-Wesley, 1968.